The Dissertation Committee for Nate F. Kohl

certifies that this is the approved version of the following dissertation:

Learning in Fractured Problems with Constructive Neural Network Algorithms

Committee:

Risto Miikkulainen, Supervisor

Dana Ballard

Benjamin Kuipers

Joydeep Ghosh

Peter Stone

Learning in Fractured Problems with Constructive Neural Network Algorithms

by

Nate F. Kohl, B.A., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2009

For Becky.

Learning in Fractured Problems with Constructive Neural Network Algorithms

Publication No. _____

Nate F. Kohl, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Risto Miikkulainen

Evolution of neural networks, or neuroevolution, has been a successful approach to many low-level control problems such as pole balancing, vehicle control, and collision warning. However, certain types of problems — such as those involving strategic decision-making — have remained difficult to solve. This dissertation proposes the hypothesis that such problems are difficult because they are fractured: The correct action varies discontinuously as the agent moves from state to state.

To evaluate this hypothesis, a method for measuring fracture using the concept of function variation of optimal policies is proposed. This metric is used to evaluate a popular neuroevolution algorithm, NEAT, empirically on a set of frac-

tured problems. The results show that (1) NEAT does not usually perform well on such problems, and (2) the reason is that NEAT does not usually generate local decision regions, which would be useful in constructing a fractured decision boundary.

To address this issue, two neuroevolution algorithms that model local decision regions are proposed: RBF-NEAT, which biases structural search by adding basis-function nodes, and Cascade-NEAT, which constrains structural search by constructing cascaded topologies. These algorithms are compared to NEAT on a set of fractured problems, demonstrating that this approach can improve performance significantly. A meta-level algorithm, SNAP-NEAT, is then developed to combine the strengths of NEAT, RBF-NEAT, and Cascade-NEAT. An evaluation in a set of benchmark problems shows that it is possible to achieve good performance even when it is not known a priori whether a problem is fractured or not. A final empirical comparison of these methods demonstrates that they can scale up to real-world tasks like keepaway and half-field soccer. These results shed new light on why constructive neuroevolution algorithms have difficulty in certain domains and illustrate how bias and constraint can be used to improve performance. Thus, this dissertation shows how neuroevolution can be scaled up from learning low-level control to learning strategic decision-making problems.

Contents

Chapter 1

Introduction

Many challenging problems that the field of artificial intelligence aims to solve involve high-level strategy. The domain of robotic soccer is an excellent example of such a problem (Figure 1.1). In this problem, a team of agents designed to play soccer against an opposing team must be able to follow a strategy, both individually and collectively, in order to defeat an organized opponent. Soccer requires agents to employ a variety of strategic behaviors – such as multi-agent collaboration and planning against adversaries – all of which are desirable traits for intelligent agents to possess. The large amount of continuous and sometimes hidden state information as well as the complex dynamics of multi-agent environments make this kind of problem quite challenging.

This dissertation aims to extend a promising reinforcement learning technique, neuroevolution, to learn strategic behavior. Through a process of empirical experimentation and analysis, the strengths and weaknesses of a state-of-the-art neuroevolution algorithm, NEAT, are evaluated. The resulting insights into why this approach has difficulty on certain problems (i.e. those where the decision boundary is fractured) are then used as inspiration for creating three new algorithms that are able to learn strategic high-level behavior.

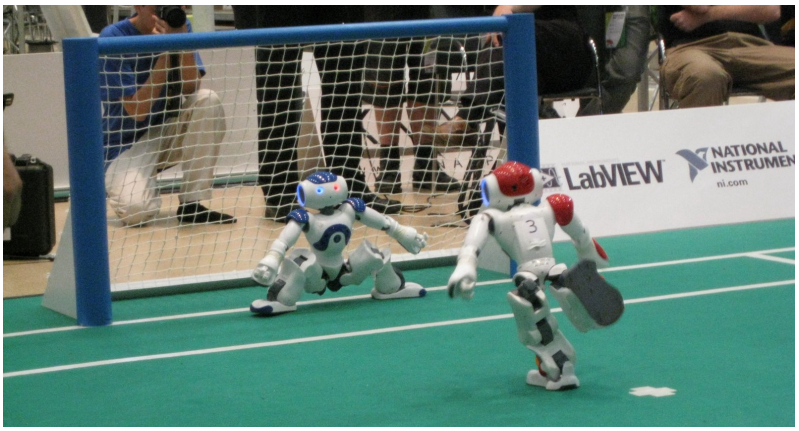


Figure 1.1: Two robots competing in a game of soccer in the RoboCup Standard Platform League. Robotic soccer is one example of a challenging multi-agent problem where employing high-level strategic behavior successfully can mean the difference between victory and defeat. (Photograph credit: Todd Hester)

1.1 Motivation

Mastering a strategic decision problem such as robotic soccer domain requires training agents to exhibit a wide array of behaviors. When near the ball, an individual player must be able to determine if it should attempt to gain control of the ball or leave it for a teammate. When in possession of the ball, a player must evaluate the position of teammates and opponents (frequently a large number of players) to determine how best to handle the ball. If a teammate assumes control of the ball, the player must decide how to position itself relative to the other players despite multiple competing objectives (e.g. saving energy for a future play versus immediately moving into an opportune spot on the field). The team as a whole must collectively make numerous decisions, from executing set plays to deciding when to press and attack versus retreating into a defensive stance. To ensure smooth behavior, all of these behaviors must be integrated into a single strategy that can be executed in a consistent manner. In addition, any differences between the opposing players (such as speed or past tactics) should be taken into account when making all of these

decisions.

The strategies required to perform well at soccer are general and are applicable to many other types of interesting problems. Being able to reason effectively with teammates in soccer is a specific example of general multi-agent strategy. An algorithm that can successfully learn to coordinate a team of soccer players could also coordinate a group of cars on the highway to maximize throughput, or could organize a troop of autonomous robots surveying a disaster area for possible survivors. Being able to predict successfully what an opposing player on the soccer field will do is an example of opponent modeling, and could be used by software bidding agents to out-maneuver other agents in online auctions. An ability to extract meaningful information from a high-dimensional stream of continuous inputs is required on a soccer field, but also is a frequent problem in real-world robotics. In short, advances that allow agents to learn high-level strategy for problems like soccer would be useful to many other areas in computer science.

1.2 Challenge

However, learning to exhibit all of the behaviors necessary to excel at high-level strategic problems like soccer has proven to be a difficult challenge. Many approaches that have been successfully employed on seemingly related problems perform poorly on problems like soccer. One example of such an approach is the neuroevolution algorithm called NEAT.

The process of evolving neural networks using genetic algorithms, or neuroevolution, is a promising approach to solving complex reinforcement learning problems. The NeuroEvolution of Augmenting Topologies (or NEAT) algorithm is one of the most recent successful neuroevolution algorithms (Stanley and Miikkulainen, 2002). While the traditional method of solving reinforcement learning problems involves the use of temporal difference methods to estimate a value function (Sutton

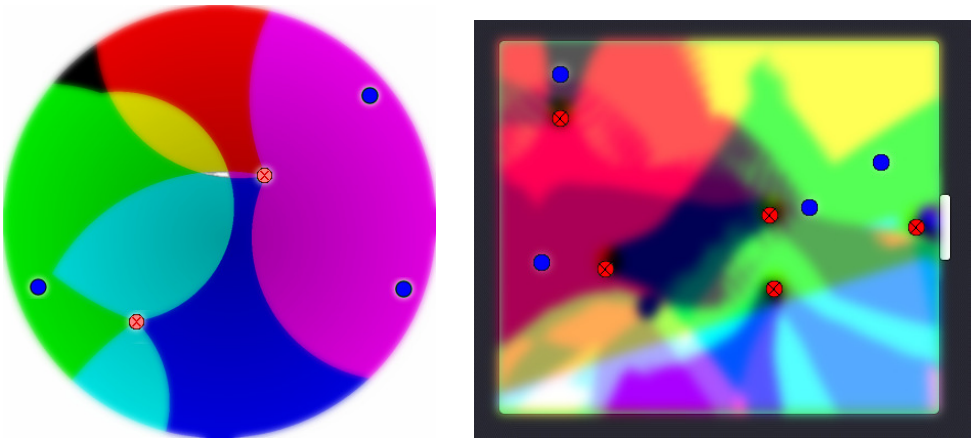
and Barto, 1998), NEAT instead relies on policy search to build a neural network that directly maps states to actions. This approach has proved to be useful on a wide variety of problems and is especially promising in challenging tasks where the state is only partially observable, such as pole balancing, vehicle control, collision warning, and character control in video games (Gomez et al., 2006, Stanley and Miikkulainen, 2002, Kohl et al., 2006, Reisinger et al., 2007, Stanley et al., 2005b, Stanley and Miikkulainen, 2004a,b). However, despite its efficacy on such low-level control problems, other types of problems such as concentric spirals classification, multiplexer, and high-level decision making in general have remained difficult for neuroevolution algorithms like NEAT to solve.

Given how effective neuroevolution has been on other problems, it is useful to understand why it has trouble performing well on problems – like soccer — that require players to exhibit high-level strategy. Any improvements that stem from such increased understanding are widely applicable to other compelling high-level strategy problems. Thus, this dissertation focuses on the following two questions:

1. Why do neuroevolution algorithms have difficulty in solving problems requiring high-level strategy, and
2. how can the performance of these algorithms be improved?

1.3 Approach

The approach taken to strategy learning in this dissertation focuses on the *fractured problem hypothesis*. This hypothesis posits that high-level strategic problems like soccer are difficult to solve because the optimal actions change abruptly and repeatedly as agents move from state to state. If learning algorithms such as NEAT have



(a) (b)

Figure 1.2: An illustration of which actions would be successful for a player with the ball at various points on the field in (a) keepaway and (b) half-field soccer. Each color represents a different set of successful actions. As players move around the field, the set of appropriate actions changes abruptly and repeatedly, giving these problems a fractured quality.

difficulty in modeling this type of fractured decision space, then strategic problems will be quite difficult to solve.

For example, Figure 1.2 shows the fractured decision spaces of two high-level strategy problems, keepaway and half-field soccer (described in detail in Chapter 8). Players on each field are represented by small circles, with teammates in blue and opponents in red with crosses. The color at each point p represents the set of successful actions from which the player with the ball at point p can choose. As the player moves around the field with the ball, the set of actions that will not immediately end the game changes frequently and discontinuously, giving these tasks a fractured quality. In addition, the nature of the fracture changes as both teammates and opponents move. The fractured problem hypothesis suggests that neuroevolution algorithms like NEAT perform poorly on such fractured problems because the evolved neural networks have difficulty representing such abrupt decision boundaries.

The first step in evaluating this hypothesis is formulating a precise definition of fracture. Using the concept of function variation, it turns out to be surprisingly simple to define and measure fracture for simple problems. If an optimal policy for a problem is known, then that policy can be treated as a continuous function and measured for variation. The resulting number is an indicator of the degree to which the problem is fractured.

Armed with a precise metric for measuring fracture, it becomes possible to examine the performance of NEAT on a variety of problems that vary in the degree to which they are fractured. Results from a set of experiments show that NEAT does indeed perform at a lower level as fracture increases, confirming the fractured problem hypothesis.

Further analysis of NEAT's performance on these fractured problems shows that NEAT does not always perform poorly on fractured problems. In rare situations it is able to find reasonable solutions. It turns out that these solutions are formed via a series of local modifications, suggesting that NEAT performs poorly on fractured problems because it has difficulty in reliably creating and manipulating local decision regions.

Fortunately, there are many algorithms from the related machine learning literature that address the problem of forming local decision regions. These approaches serve as inspiration for two new neuroevolution algorithms based on NEAT: RBF-NEAT, which augments the standard NEAT algorithm with the ability to add basis function nodes to network topologies, and Cascade-NEAT, which constrains the growth of network topologies to a specific and regular pattern. These two algorithms are designed to harness the power of bias and constraint in forming local decision regions to solve fractured problems.

An empirical comparison of RBF-NEAT and Cascade-NEAT to the standard NEAT algorithm shows that it is possible to increase performance on fractured

problems significantly by employing such bias and constraint. However, further experimentation shows that NEAT still performs the best on traditional benchmark problems like pole-balancing.

Therefore, it is useful to incorporate RBF-NEAT, Cascade-NEAT, and regular NEAT into a single algorithm that can take advantage of the strengths of all three approaches. SNAP-NEAT is introduced as a modification of a popular adaptive operator selection algorithm (Thierens, 2005) and is empirically shown to work well on both fractured and unfractured problems. Further empirical evaluation shows that RBF-NEAT, Cascade-NEAT, and SNAP-NEAT work well not only on a benchmark suite of fractured problems but also on the challenging high-level decision problems of keepaway and half-field soccer.

1.4 Outline

This dissertation is organized as follows:

Chapter 2 begins with an overview of the neuroevolution literature, describing both fixed topology and constructive approaches to building neural networks. One particularly promising constructive neuroevolution algorithm, NEAT, is described in detail, and serves as a basis for experimentation for the rest of the dissertation.

Chapter 3 introduces a quantitative metric based on the concept of function variation for measuring the degree to which a problem is fractured. This method of measuring fracture is simple and relatively straightforward to calculate. Details about the limitations of and assumptions behind this approach to measuring problem difficulty are also discussed.

Chapter 4 evaluates empirically the hypothesis that NEAT has difficulty on fractured problems. Several different problems are introduced that vary in the degree to which they are fractured. NEAT's performance on these problems is evaluated

and its failure modes are analyzed, which leads to the idea that an ability to create local decision regions is crucial in solving fractured problems.

Chapter 5 reviews the machine learning literature for related approaches to defining local decision regions. While there are many algorithms, two in particular seem particularly amenable to the neuroevolution approach: RBF nodes and the cascade architecture. Based on these ideas, two new neuroevolution algorithms are introduced and described in detail.

Chapter 6 compares the performance of these two new approaches to the standard NEAT algorithm on the fractured problems from Chapter 4. The new algorithms are much more effective at learning in fractured problems than NEAT. However, NEAT still yields the best performance on problems that can be solved with small, recurrent networks, such as double pole-balancing.

Chapter 7 introduces an algorithm that combines the strengths of all three neuroevolution algorithms. This combined algorithm, SNAP-NEAT, is evaluated empirically on a variety of problems, both fractured and unfractured, and is shown to be competitive in all scenarios.

Chapter 8 expands this empirical evaluation by comparing NEAT to RBF-NEAT, Cascade-NEAT, SNAP-NEAT, as well as other standard learning approaches like ESP and SARSA on two challenging high-level strategy problems. Although the complex nature of these problems limits the analysis, the neuroevolution algorithms are found to be better than the other approaches.

Chapter 9 discusses the results presented in previous chapters, describing the assumptions behind the work in this dissertation and the kinds of problems and algorithms (such as value-function reinforcement learning or supervised machine learning) to which this kind of analysis might be expanded. Directions for future work are also explored, including an examination of how different methods of network construction might be integrated into SNAP-NEAT.

Chapter 2

Learning with Neuroevolution

Neuroevolution algorithms use some flavor of evolutionary search to generate neural network solutions to reinforcement learning problems. They represent a powerful set of learning algorithms that have been shown to perform well on a variety of problems (Gomez et al., 2006, Stanley and Miikkulainen, 2002, Kohl et al., 2006, Reisinger et al., 2007, Stanley et al., 2005b, Stanley and Miikkulainen, 2004a,b). This chapter reviews the recent development of fixed-topology and constructive neuroevolution algorithms. In addition, details are provided about one of the more promising approaches, NEAT (Stanley and Miikkulainen, 2002), that will serve as a focus of investigation for this dissertation.

2.1 Fixed Topology Algorithms

Neuroevolution algorithms are frequently divided into two groups: those that optimize the weights of a fixed-topology network, and those that evolve both the network topology and weights. Most of the early work in neuroevolution dealt with fixed-topology algorithms (Gomez and Miikkulainen, 1999, Moriarty and Miikkulainen, 1996, Saravanan and Fogel, 1995, Whitley et al., 1993, Wieland, 1991). This work

was driven by the simplicity of dealing with a single network topology and theoretical results showing that a neural network with a single hidden layer of nodes could approximate any function, given enough nodes (Hornik et al., 1989).

One early approach to neuroevolution was the Symbiotic, Adaptive Neuroevolution (or SANE) algorithm (Moriarty and Miikkulainen, 1996). Instead of evolving complete networks, SANE evolves populations of neurons that are combined together to form whole networks. Allowing neurons to diversify into separate populations is an effective way to maintain diversity in the overall population, as networks formed with many of the same neurons are unable to specialize to perform specific tasks.

One popular successor to SANE, Enforced Sub-Populations (or ESP), has been used to solve a variety of difficult reinforcement learning problems such as multi-agent coordination and finless rocket control (Gomez, 2003, Yong and Miikkulainen, 2007). ESP improves upon SANE by explicitly dividing an evolving population into separate sub-populations, one for each neuron in the evolving topology. Members from each population are combined together to form a complete network, which can be evaluated in the target task. Credit from this performance is then divided between the neurons that contributed to the network. Since recombination between neurons is restricted to occur only within sub-populations, each sub-population is encouraged to specialize into a good sub-function for the network as a whole. This division of functionality has proven to be quite effective at evolving fixed-topology networks (Gomez and Miikkulainen, 2003, 2004).

Another recent approach to fixed-topology neuroevolution is the Covariance Matrix Adaptation (or CMA-ES) algorithm, which uses a type of Evolutionary Strategies algorithm to determine network weights. As its name implies, CMA-ES tracks changes between network weights and fitness to update a weight mutation distribution stored in a covariance matrix, and has been used successfully on several

benchmark problems (Igel, 2003).

However, there are certain limits associated with fixed-topology algorithms. Chief among those is the issue of choosing an appropriate topology for learning a priori. Even assuming that the general class of network topology is known (i.e. number of hidden nodes, hidden layers, recurrent layers, and the associated connectivity between nodes) there is no clear procedure to choose network size. Networks that are too large have extra weights, each of which adds an extra dimension of search. On the other hand, networks that are too small may be unable to represent solutions of a certain level of complexity, which can limit the algorithm artificially.

2.2 Constructive Algorithms

Neuroevolution algorithms that evolve both topology and weights (so-called constructive neural network algorithms, or TWEANNs, i.e. topology and weight evolving artificial neural network algorithms) were created to address this problem, and can be divided into two groups: those that directly the neural network (Dasgupta and McGregor, 1992, Pujol and Poli, 1997, Angeline et al., 1993, Yao, 1999), and those that use a method that indirectly specifies how the network should be constructed (Gruau et al., 1996, Hornby and Pollack, 2002, Stanley et al., 2009).

The simplest approach to constructive neuroevolution requires that the learning algorithm defines each part of an evolving neural network explicitly and directly. An abstract example of this approach is an algorithm that maintains a variable-length list of pieces of which an evolving network is composed. New topology can be added to the network by adding elements to the list. This list of network components can vary, and can be represented e.g. by a binary encoding (Dasgupta and McGregor, 1992) or a graph encoding (Pujol and Poli, 1997). Other aspects of the underlying genetic algorithm may also be modified, such as using crossover or a non-mating reproduction strategy (Angeline et al., 1993).

Alternatively, a network can be defined as the end result of some input data being fed through a transformative algorithm. Such encodings of networks are typically labeled "indirect", and posses the ability to generate large structures with symmetry and repeated motifs easily (Stanley and Miikkulainen, 2003). These approaches, inspired by cell division (Gruau et al., 1996), L-systems (Hornby and Pollack, 2002), and multiple levels of indirection (e.g. the HyperNEAT algorithm; Stanley et al., 2009) are intuitively appealing, but have many obstacles to overcome and have not yet met with much practical success on traditional learning benchmarks.

While many of these approaches to neuroevolution been successful, they struggle to address the difficulties in evolving both topology and weights simultaneously. In particular, one challenging problem faced by constructive algorithms is exploring the high-dimensional space of network topologies quickly and efficiently.

2.3 NEAT

One of the most popular constructive neural network algorithms is Neuroevolution of Augmenting Topologies (NEAT; Stanley and Miikkulainen, 2002). NEAT uses *complexification* to search through the high-dimensional space of network topologies: It starts with simple networks and expands the search space only when beneficial, allowing it to find significantly more complex controllers than fixed-topology evolution can. Because NEAT starts searching in relatively low-dimensional spaces, it avoids wasting time looking for overly complex solutions, making this approach an attractive method for evolving neural networks in complex tasks.

NEAT's ability to discover arbitrary network topologies and its past successes on a variety of reinforcement learning problems make it a natural focus for further experimentation. It would be worthwhile for practitioners in the field to better understand when NEAT works and when it fails. Furthermore, given NEAT's competitive results in various reinforcement learning benchmarks, a deeper under-

standing could be useful to the machine learning field at large. In this section, the NEAT method is briefly reviewed; see (Stanley and Miikkulainen, 2002, 2004a) for more detailed descriptions.

NEAT is based on three key ideas. First, evolving network structure requires a flexible genetic encoding that allows two networks with arbitrary topology to be recombined. Each genome in NEAT includes a list of connection genes, each of which refers to two node genes being connected. Each connection gene specifies the in-node, the out-node, the weight of the connection, whether or not the connection gene is expressed (an enable bit), and an innovation number, which allows finding corresponding genes during crossover. Mutation can change both connection weights and network structures. Connection weights are mutated in a manner similar to any neuroevolution system. Structural mutations, which allow complexity to increase, either add a new connection or a new node to the network. Through mutation, genomes of varying sizes are created, sometimes with completely different connections specified at the same positions.

Since NEAT works with many different network topologies at any given time, one interesting problem that it faces is how to perform meaningful crossover on networks with different topologies. This issue — known as the competing conventions problem — is solved by NEAT by using innovation numbers to "line up" genes during crossover. This so-called artificial synapsis (named after a similar process that occurs in biological organisms) allows the crossover operator to select between meaningful parts of network topology, without expensive topological analysis. Each unique gene in the population is assigned a unique innovation number when the gene is created, and the numbers are inherited during crossover. An example of this process of artificial synapsis is shown in Figure 2.1. Genomes of different organizations and sizes stay compatible throughout evolution, and the problem of matching different topologies (Radcliffe, 1993) is essentially avoided.

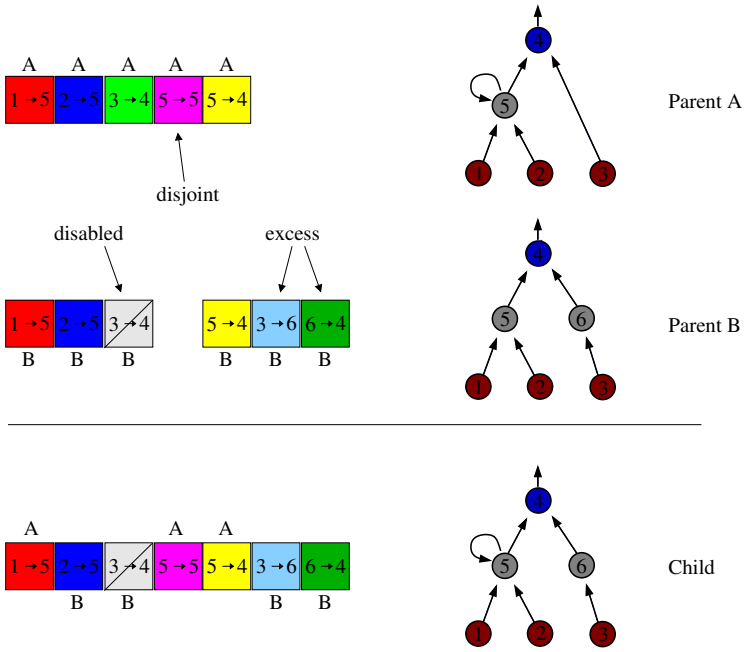


Figure 2.1: An example of how NEAT employs artificial synapsis via innovation numbers, indicated by the color of each gene in this figure. By providing a principled mechanism to align genetic information between two genomes, NEAT is able to perform meaningful crossover between networks with different topologies.

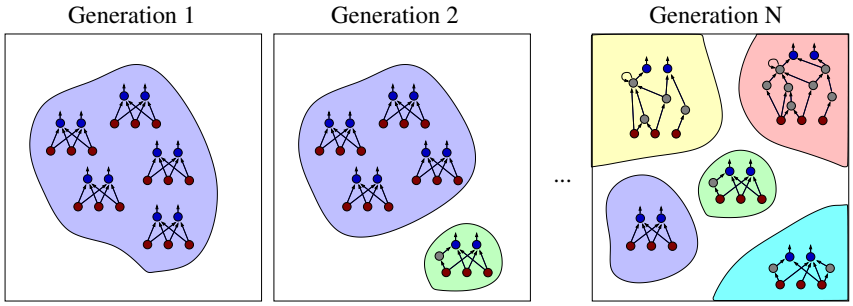


Figure 2.2: An example of complexification in NEAT. An initial population of small networks gradually speciates into a more diverse population. This process allows NEAT to search efficiently in the high-dimensional space of network topologies.

Second, NEAT speciates the population so that individuals compete primarily within their own niches instead of with the population at large. This way, topological innovations are protected and have time to optimize their structure before they have to compete with other niches in the population. The reproduction mechanism for NEAT is explicit fitness sharing (Goldberg and Richardson, 1987), where organisms in the same species must share the fitness of their niche, preventing any one species from taking over the population.

Third, unlike other systems that evolve network topologies and weights (Gruau et al., 1996, Yao, 1999), NEAT begins with a uniform population of simple networks with no hidden nodes. New structure is introduced incrementally as structural mutations occur, and the only structures that survive are those that are found to be useful through fitness evaluations. This process of starting small and incrementally adding structure is depicted in Figure 2.2. In this manner, NEAT searches through a minimal number of weight dimensions and finds the appropriate level of complexity for the problem.

This approach is highly effective: NEAT has outperformed other neuroevolution methods on complex control tasks like double pole balancing (Stanley and Miikkulainen, 2002) and robotic strategy-learning (Stanley and Miikkulainen, 2004a).

However, it has turned out to be surprisingly difficult to get NEAT to perform well in other problems, such as concentric spirals, multiplexer, and high-level strategy problems in general. This issue is the research opportunity that is pursued in this dissertation.

2.4 Learning in Fractured Problems

The three principles described above allow NEAT to search quickly and efficiently through the space of possible network topologies to find the right neural network for the task at hand. However, NEAT is limited to small, incremental changes in the network structure. While such mutations are useful when building relatively small networks, tasks that require complicated or repeated internal structure are difficult for NEAT. Furthermore, any small mutations that are made to network structure can potentially have a global impact on network output. If a task requires local adjustments to network output, NEAT's performance may suffer.

The problems to which NEAT has been applied so far can be solved by relatively small neural networks with continuous output, so these potential drawbacks may not have been an issue in the past. In fact, since NEAT starts with a minimal topology and uses small mutations to tweak network architecture, it is unusually well suited to such problems. The double pole-balancing reinforcement learning benchmark problem is a prime example of this type of problem. NEAT has been shown to work extremely well on double pole-balancing, outperforming other neuroevolution algorithms and standard reinforcement learning algorithms (Stanley and Miikkulainen, 2004a). The types of solutions that NEAT discovers (shown in Figure 2.3) are surprisingly compact, demonstrating the power of recurrent neural networks as a representation for this type of continuous control problem.

On the other hand, not all problems have optimal policies that can be represented by small, recurrent neural networks. Although publishing biases make it

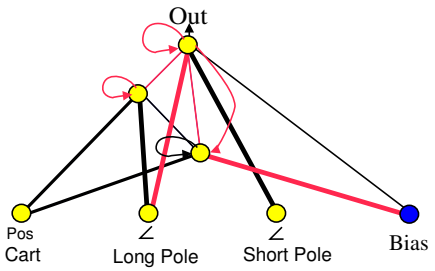


Figure 2.3: A surprisingly small solution that was evolved by NEAT to solve the non-Markov double pole balancing problem. Shared recurrent connections between the two poles allow the network to compute the velocity of the poles, allowing NEAT to generate a parsimonious solution to this problem.

difficult to find negative results in the literature, it is apparent that there are many types of interesting problems that algorithms like NEAT cannot solve. While many of the successful applications of NEAT focus on low-level reasoning and continuous control problems, it is rare to find examples of NEAT successfully being used to evolve high-level strategic behavior. In addition, informal experimentation has shown that it is difficult for NEAT to solve problems like SAT, multiplexer, or the concentric spirals classification task.

What makes these problems different from those on which NEAT and other neuroevolution algorithms have done so well? This dissertation explores the hypothesis that these problems share a common property: They possess a "fractured" decision space, loosely defined as a *space where adjacent states require radically different actions*. Furthermore, it is posited that constructive neuroevolution algorithms like NEAT have difficulty in assembling the structure necessary to model this type of input.

In order to build the next generation of neuroevolution algorithms, it will be necessary to look beyond the current success stories and understand why algorithms like NEAT perform poorly on certain classes of problems. The concept of fracture is one possible explanation for this behavior, and will serve both as a focal point for

analysis of NEAT and as inspiration for improved algorithms in this dissertation.

2.5 Conclusion

There have been many interesting and powerful approaches to evolving neural networks for reinforcement learning problems. Among the constructive algorithms, the NEAT algorithm has been shown to be surprisingly versatile at evolving solutions to low-level control problems.

However, success at evolving high-level behavior has proven to be elusive, possibly because of NEAT's difficulty in modeling the fractured state spaces of high-level decision problems. The next chapter examines this concept of fracture more closely and describes one method for measuring it explicitly.

Chapter 3

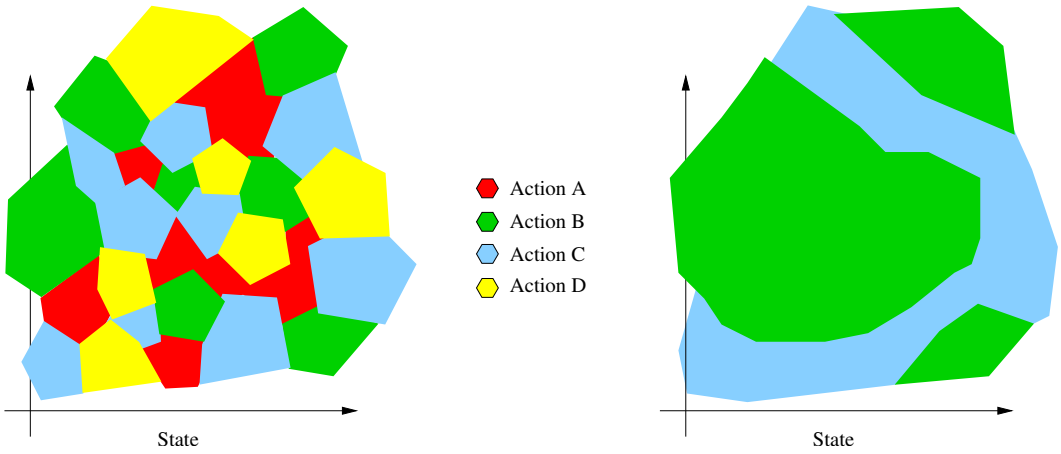
Fractured Problems: Definition

In this chapter, the description of fracture is elaborated and the concept of function variation is introduced as a way to more precisely quantify problem fracture. The next chapter then describes several experiments that test the hypothesis that the difficulty neuroevolution has with fractured problems stems from an inability to generate networks with an appropriate amount of variation.

3.1 Fractured Problems and Function Variation

For many problems (such as the typical control problems or the standard reinforcement learning benchmarks), the correct action for one state is similar to the correct action for neighboring states, varying smoothly and infrequently. In contrast, for a fractured problem, the correct action changes repeatedly and discontinuously as the agent moves from state to state. Figure 3.1 shows simple examples of a fractured and unfractured two-dimensional state space.

Clearly, the choice of state variables could change many aspects of a given problem, including the degree to which it is fractured. For example, solving the concentric spirals problem becomes much easier if the state space is represented in



(a) (b)

Figure 3.1: A simple example of a 2-d state-action space that is (a) fractured and (b) unfractured. In (a), the correct actions vary frequently and discontinuously as an agent moves through the state space. If a learning algorithm cannot represent these abrupt changes, its performance will be limited.

polar coordinates instead of cartesian coordinates. In this dissertation, a problem is considered a "black box" that already has associated states and actions. In other words, it is assumed that the definition of a problem includes a choice of inputs and outputs, and the goal of the agent is to learn given those constraints. Any definition of fracture then applies to the entire definition of the problem, including representation details such as input and output.

The definition of fracture provided above, while intuitive, is not very precise. More formal definitions of difficulty have been proposed for learning problems, including Minimum Description Length (Barron et al., 1998, Chaitin, 1975), Kolmogorov complexity (Kolmogorov, 1965, Li and Vitanyi, 1993), and Vapnik-Chervonenkis (VC) dimension (Vapnik and Chervonenkis, 1971). Unfortunately, these metrics are often more suited to a theoretical analysis than they are to practical usage. For example, Kolmogorov complexity is a measure of complexity that depends on the computational resources required to specify an object — which

sounds promising for measuring problem fracture — but it has been shown to be uncomputable in practice (Maciejowskia, 1979).

One method of measuring problem fracture is to consider the degree to which solutions to a problem are fractured. VC dimension at first appears promising for this approach, since it describes the ability of a possible solution to "shatter" a set of randomly-labeled training examples into distinct groups. Unfortunately, VC dimension is a very general way of measuring the capabilities of a model, and does not apply to a specific problem. Also, analyzing VC dimension for neural networks is difficult; while results exist for single-layer networks, it is much more difficult to analyze the networks with arbitrary (and possibly recurrent) connectivity that constructive neuroevolution algorithms generate (Mitchell, 1997). These limitations of VC theory render it inappropriate for measuring problem fracture.

Of more interest is the work of Ho and Basu, who surveyed a variety of complexity metrics for supervised classification problems and found a significant difference between random classification problems and those drawn from real-world datasets (Ho and Basu, 2002). In terms of measuring problem fracture, the most promising of these metrics is a gauge of the linearity of the decision boundary between two classes of data. However, these metrics are tied to a two-class supervised learning setting, which makes them less useful in a reinforcement learning setting, where the goal can involve learning a continuous mapping from states to actions.

Assuming that the goal of the search algorithm is to find an optimal policy for the problem, this work adopts the approach of measuring problem fracture directly by considering how much the actions of optimal policies for the problem change from state to state. In this way of thinking, a fractured problem is characterized as a problem where the policies that are being sought repeatedly yield different actions as the agent moves from state to state. Compared to the alternatives described above, this definition of problem fracture is easy to compute, because it turns out

to be surprisingly simple to measure how much policies change over a fixed area.

Of course, this definition of problem fracture explicitly ties fracture to how much optimal policies change. Intuitively, a problem may be considered difficult to solve (i.e. to find an optimal policy) if the policies being searched for have this fractured property. While this definition of fracture is intuitive and relatively easy to understand, it does ignore a variety of factors, including reward function and operators used in the learning process. Regardless, this simple metric has proven to be quite useful in analyzing and improving neuroevolution algorithms like NEAT, as well be shown below. Discussion about the relative merits of this approach will be revisited in Chapter 9.

Estimating problem fracture depends on measuring how the actions of optimal policies change from state to state. The next section describes how this measurement can be made by treating policies as functions and measuring how much the functions change using the concept of total variation.

3.2 Total Variation of a Function

One metric that measures how much a function (or policy) changes over a certain interval is known as the *total variation* of a function (Leonov, 1998, Vitushkin, 1955). This section provides a technical description of multidimensional variation (adapted from (Leonov, 1998)) followed by several illustrations of how variation can be computed.

Consider an N-dimensional rectangular parallelepiped $B = B_a^b = B_{b_1 \ldots b_N}^{a_1 \ldots a_N} = \{x \in \Re^N : a_i \leq x_i \leq b_i, a_i < b_i, i = 1, \ldots, N\}$ and a function over this parallelepiped, $z(x_1, \ldots, x_N)$, whose variation is to be measured. From (Leonov, 1998, Bochner, 1959, Kamke, 1956, Shilov and Gurevich, 1967), the "N-dimensional quasivolume" σ_N for z over a sub-parallelepiped B_α^β of B is defined as

$$\sigma_N(B_\alpha^\beta) = \sum_{v_1=0}^{1} \cdots \sum_{v_N=0}^{1} (-1)^{v_1+\ldots+v_N} z[\beta_1+v_1(\alpha_1-\beta_1), \ldots, \beta_N+v_N(\alpha_N-\beta_N)] \quad (3.1)$$

Now consider a partitioning of B into a set of sub-parallelepipeds $\Pi = \{B_j\}_{j=1}^n$ where none of the individual sub-parallelepipeds B_j intersect and $B_1 + \ldots + B_n = B$. Let P be the set of all such partitions for all n. The N-dimensional variation (or Vitali variation) of the function z in the parallelepiped B is

$$V_N(z, B) = \sup_{\Pi} \left\{ \sum_{j=1}^{n} |\sigma_N(B_j)| : \Pi = \{B_j\}_{j=1}^n \in P \right\} \quad (3.2)$$

Next, consider all of B's m-dimensional coordinate faces $B_{i_1,\ldots,i_m}$ for $1 \leq m \leq N-1$ that pass through the point $a \in B$ and are parallel to the axes $x_{i_1}, \ldots, x_{i_m}$ where $1 \leq i_1 < \ldots < i_m \leq N$. For convienence, mark all of the m-dimensional faces of the form $B_{i_1,\ldots,i_m}$ by a number r ($1 \leq r \leq \binom{N}{m} = N_m$). Each such face will be denoted by $B_r^{(m)}$.

Definition: The *total variation of the function z in the parallelepiped B* is the number

$$V(z, B) = \sum_{m=1}^{N-1} \left\{ \sum_{r=1}^{N_m} V_m(z, B_r^{(m)}) \right\} + V_N(z, B) \quad (3.3)$$

Several illustrative examples of this variation calculation follow, starting with the variation calculation in one dimension. The variation of a 1-d function $z(x_1)$ over the range $a_1 \leq x_1 \leq b_1$ is simply the sum of the absolute value of the differences between adjacent values of z between a_1 and b_1. When $N = 1$, the variation of z over the interval B, $V(z, B)$, effectively becomes $V_1(z, B)$, which is computed by the summation in Equation 3.2. For example, Figure 3.2 shows a function z that has been divided into five sections inside the interval $[a_1, b_1]$. The differences between adjacent points (each computed by Equation 3.1 and shown in red in Figure 3.2)

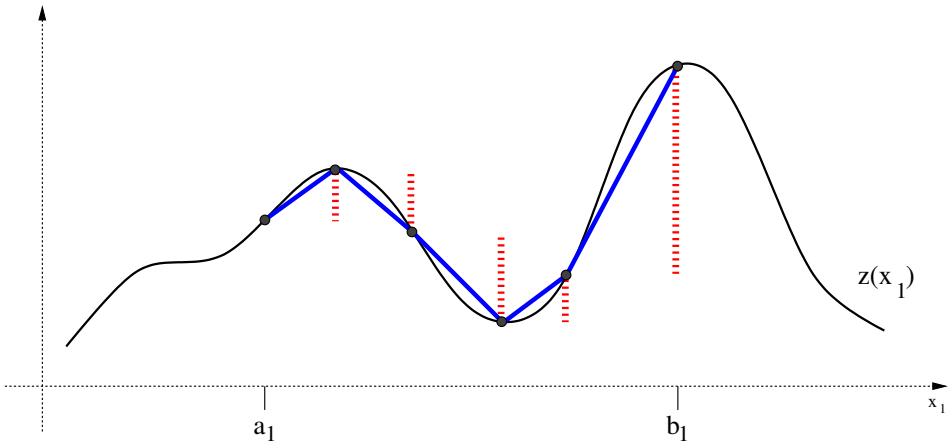


Figure 3.2: An example of how the variation of a 1-d function is computed. The absolute value of the differences between adjacent points on the function (shown in red) are summed together to produce an estimate of the total variation.

would be added together to determine the variation for z on the parallelepiped $B = B_{a_1}^{b_1}$, which is just the 1-d interval $[a_1, b_1]$.

In Figure 3.2, the 1-d parallelepiped B (or the interval $[a_1, b_1]$) is divided into five sections by six points. Clearly, a different selection of points could produce a different estimate of variation. For example, if the variation calculation only used the first and last points in the interval, then the middle two "bumps" of the function would be skipped over, producing a lower variation. The choice of an appropriate set of points (referred to above as a partition Π of B) is dealt with in Equation 3.2. To compute the $V_N(z, B)$, a partition Π of B should be chosen such that it maximizes $V_N(z, B)$. It is easy to see that as the discretization of the partition becomes increasingly fine, the variation will not decrease. In fact, when the discretization of the partition becomes infinitely small, the calculation of $V_N(z, B)$ in the 1-d case turns into

$$V_1(z, B_{a_1}^{b_1}) = \int_{a_1}^{b_1} |z'(x)| dx \tag{3.4}$$

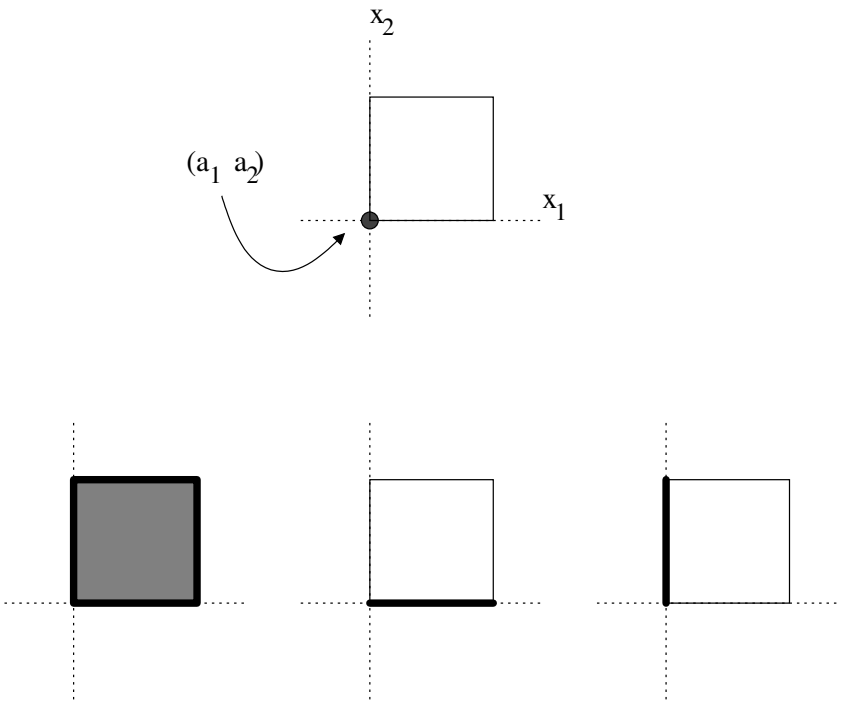


Figure 3.3: The three faces of a 2-d parallelepiped. Measuring variation on each face is meant to capture how the function changes in different directions.

Infinitely-fine partitionings of B are fine for mathematicians, but practically speaking, computational resources will limit the degree to which it is possible to discretize B. In this dissertation, the finest possible discretization $\hat{\Pi}$ of B is chosen given the limited computational resources available. This choice means that only one partition $\hat{\Pi}$ is considered, and the supremum in Equation 3.2 is effectively ignored.

The computation for multidimensional variation is more involved than the 1-d case. To compute the variation for a 2-d function $z(x_1, x_2)$ on the parallelepiped $B = B_{a_1,a_2}^{b_1,b_2}$, three different terms are computed and summed together. Each of these terms is meant to measure the variation in a specific direction on B, and corresponds to a "face" of B (shown in Figure 3.3):

- $B_1^{(1)}$: the first 1-d face of B, variation of z as x_1 changes

- $B_2^{(1)}$: the second 1-d face of B, variation of z as x_2 changes
- B: the only 2-d face of B is B itself, variation of z as both x_1 and x_2 change

To compute the variations for the two 1-d faces of B, $V_1(z, B_1^{(1)})$ and $V_1(z, B_2^{(1)})$, a calculation very similar to the one described above can be used: The variation is simply the sum of the absolute values of the differences between adjacent values of the function. Each difference between adjacent points α and β is $\sigma_N(B_\alpha^\beta)$, represented by Equation 3.1. The 2-d version of Equation 3.1 involves measuring four points, instead of two. For example, when $N = 2$, the qausivolumes of the function z over the parallelepipeds $B_{\alpha_1,\alpha_2}^{\beta_1,\beta_2}$, $B_{\alpha_1}^{\beta_1}$, and $B_{\alpha_2}^{\beta_2}$ are

$$\sigma_2(B_{\alpha_1,\alpha_2}^{\beta_1,\beta_2}) = z(\beta_1, \beta_2) - z(\beta_1, \alpha_2) - z(\alpha_1, \beta_2) + z(\alpha_1, \alpha_2) \tag{3.5}$$

$$\sigma_1(B_{\alpha_1}^{\beta_1}) = z(\beta_1, a_2) - z(\alpha_1, a_2) \tag{3.6}$$

$$\sigma_1(B_{\alpha_2}^{\beta_2}) = z(a_1, \beta_2) - z(a_1, \alpha_2) \tag{3.7}$$

It should be noted that there are actually four 1-d faces of the 2-d parallelepiped B, but only two of the faces are used. The two faces that are used are those that are on the "lower" edge of B, denoted by those faces that pass through the point $a \in B$.

Figure 3.4 illustrates another example of how multidimensional variation is computed, now using the seven chosen faces of a 3-d parallelepiped B. Again, note that only the lower faces of B are considered for this calculation of variation.

The next section continues this discussion of function variation with a description of how total variation can be measured in the context of neuroevolution.

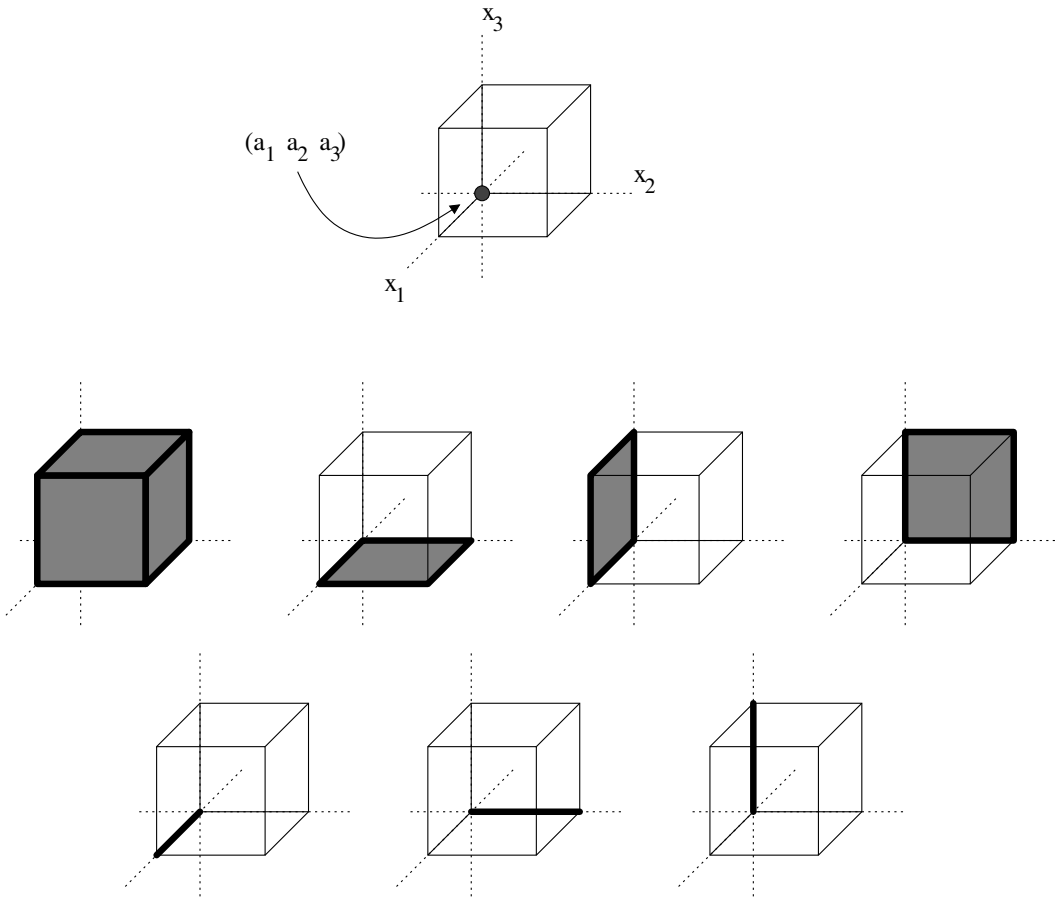


Figure 3.4: The seven chosen 1-d, 2-d, and 3-d faces of a 3-d parallelepiped. Note that all chosen faces pass through the point (a_1, a_2, a_3).

3.3 Fracture, Variation, and Neural Networks

A neural network produced by a neuroevolution algorithm can be thought of as a function that maps states to actions. Because the variation calculation does not care what form the function takes — it only requires input and output pairs from the function — it is straightforward to calculate the variation of a neural network.

The first step in measuring the variation of a neural network is selecting a parallelepiped P of the input space over which variation will be measured. In a reinforcement learning setting, the inputs have usually already been truncated or scaled to a certain range, effectively defining P. For example, an agent controlling a racecar might receive an angular input describing the location of the nearest opponent. This input can be scaled into the range $[-\pi, \pi]$, defining P for that dimension. Different dimensions of P can have different bounds.

The next step involves quantizing P into some partition Π. As described above, the ideal partition $\Pi_{optimal}$ contains infinitely small slices of P. However, a finite amount of computation limits how finely P can be discretized. Practically speaking, P is quantized into $\hat{\Pi}$, which is the finest uniform discretization of P that is possible given the computational resources that are available.

The definition of P and $\hat{\Pi}$ determine a finite set of points from the input space. The output of the neural network is then measured and stored for each of these input points. After measuring these values, a series of summations over each possible combination of dimensions of the input space (described by Equations 3.1, 3.2, and 3.3) determines the variation of the network. For networks with multiple outputs, this work assumes that the total variation of the network is the average of the multiple independent variation calculations for each output.

It should be noted that this definition of variation relies on the network representing a function, as defined in a mathematical sense. In some applications of neuroevolution, evolved networks are not functions in the strictest sense; they

map states to actions, but the experimenter does not reset node activation levels between successive states. Evolved networks used in this manner are less similar to state-action functions and more akin to dynamical systems with internal state over time that happen to pick actions. With such networks, it is not meaningful to simply query a network for its chosen action given a single state. Instead, the network must be evaluated over a set of states, starting from a specific initial state, while maintaining activation levels of individual nodes in the network across state transitions. Maintaining this continuity between states can be used to great advantage during learning, especially for non-Markov problems. For example, in an application to a non-Markov pole-balancing task, an evolved network was found to use recurrent connections and continuous state activations (without resets) to compute the derivative of the pole angle efficiently (Stanley and Miikkulainen, 2002). This information about the direction of pole movement proved useful in solving the task quickly with a small network (Figure 2.3).

However, it is difficult to measure the variation of a network that is not used in a functional sense. Instead of simply querying the network for its output at a given state, the entire succession of states that lead up to the state in question must be queried in order – and it is still not clear that such an approach would yield appropriate values for a variation computation. Because of this restriction, variation will be measured in this dissertation only on Markov problems. Fortunately, there are many interesting problems that are Markov or that can be made Markov with additional state variables, and such analysis leads to useful insight about learning algorithms in general.

It is important to note that the Markov restriction applies only to variation analysis, not to algorithms developed based on this analysis. In fact, while the variation analysis in Chapter 4 focuses on Markov problems, the analysis results in three learning algorithms that are then applied to non-Markov problems in Section 7.3.1.

Interestingly, the complex, recurrent topologies that constructive neuroevolution algorithms produce are useful in Markov problems as well. When such a network is used to produce an action for a given state, it starts from a uniform un-activated state where only the input nodes are producing output. It is then activated κ times, where each activation allows values from the input nodes to propagate one level deeper into the network. The input nodes maintain their output over all κ activations. In addition, during the first activation, values from the input nodes are propagated through the network until all output nodes have received some input value. While such networks do not aggregate information across states, this repeated activation scheme does allow recurrent connections and values delayed during propagation to be used as part of the computation of an action for a state. In this manner, complex network topologies with resets can be used most effectively in Markov problems, and without resets in non-Markov problems, as is done in this dissertation.

3.4 Conclusion

One of the most straightforward ways to determine fracture in a problem is to measure the amount of functional variation in optimal policies. This approach is relatively simple, well-founded mathematically, and can be used on any functional form (e.g. neural networks).

Using the procedure described by Equation 3.3, it is possible to evaluate the variation of neural networks produced by a learning algorithm. In the experiments described in the next chapter, this calculation is used to measure the variation of known optimal policies for problems as well as the variation of networks produced by NEAT.

Chapter 4

Fractured Problems: Analysis

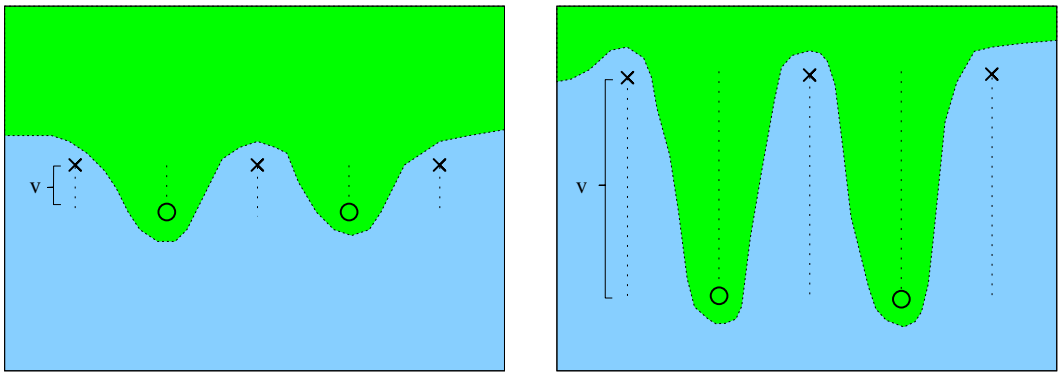
NEAT has been shown to work well on a variety of challenging low-level control problems, but has not performed as well on certain high-level problems. The hypothesis presented in Chapter 2 posits that this lack of performance arises from the inherent difficulties in assembling neural topologies that model fractured decision boundaries. This chapter evaluates this hypothesis empirically and describes several experiments that suggest how to improve NEAT's performance.

4.1 Evaluating NEAT on Fractured Problems

In order to test the fractured-problem hypothesis, NEAT's performance was evaluated on several problems that featured different amounts of variation. The goal was to measure the effect on NEAT's performance as the amount of variation required to solve the problem increased.

4.1.1 N-Point classification

The first problem examined was a simple N-point classification task. The goal of this problem is to properly classify each of a set of N alternating points into one of



(a) (b)

Figure 4.1: Examples of solutions to the N-Points problem when $N = 5$ and the separation between the two classes of points is (a) $\upsilon = 0.1$ and (b) $\upsilon = 0.8$. As υ increases, the two classes of points move further away from each other, and the minimal variation required to describe the boundary between the two classes increases.

two groups. A variety of different versions of this problem were created to examine how different amounts of variation in optimal policies impact NEAT's performance.

For each of four values of $N = 3, 5, 7, 10$, 13 different problems were created, resulting in a total of 52 problems. Within each group of 13, the two classes of alternating points were separated by amounts varying from $\upsilon = 0.01$ to 1.0. When υ was low, the optimal policy for distinguishing the two groups required very little variation. As υ increased, so did the amount of variation required for an optimal policy. For example, Figure 4.1 shows two of the 13 problems when $N = 5$, along with examples of optimal policies for each problem. When the two classes of points are relatively close together, the decision boundary between the two classes can be relatively smooth. As the two classes are moved farther apart, the boundary becomes increasingly non-linear, which increases the minimal variation required to describe the boundary.

Each network was evaluated on a series of inputs, each representing one of the N points. Network activation was reset between successive inputs. The

correct output for the network depended on both the class to which the current point belonged and the separation parameter υ. When υ was small, there was a large range of values that the network could produce that would yield a correct classification. As υ increased, that window shrank, such that when $\upsilon = 1.0$, the only correct values that a network could produce were 0 if the point belonged to the first class and 1 if it belonged to the second class.

The minimal amount of variation required to solve an instance of the N-Point classification problem is relatively straightforward to compute. In order to separate the N alternating points into two groups, a function must alternate between producing values at least as large as $\upsilon/2$ and at least as small as $-\upsilon/2$. Since there are $N-1$ gaps between adjacent points that the function must alternate over, the minimum amount of variation required to properly classify all N points is $(N-1)\upsilon$.

After being presented with all N input points, the score for a network was defined to be $10 - \chi$, where χ was the number of misclassified points. The results for all 52 versions of this problem are shown in Figure 4.2.

The different values of the υ parameter of the N-Points problems allows the effect of changing variation to be clearly seen. As υ increases for a given N, very little about the problem or fitness function changes except the amount of variation required to solve the problem. As this required variation increases, NEAT's performance goes down. Within each group of N, the differences in score for the lowest, middle, and highest values of υ are all significant with $p = 0.99$.

The amount of variation required to solve the problem also increases as the number of points N increases. It is interesting to note that each additional pair of points adds another "inflection point" that the optimal policy must represent. The concept of the amount of inflection of a policy represents a possible complement to policy variation, since inflection and variation can vary independently of each other and could each describe different aspects of policy complexity. Note, however,

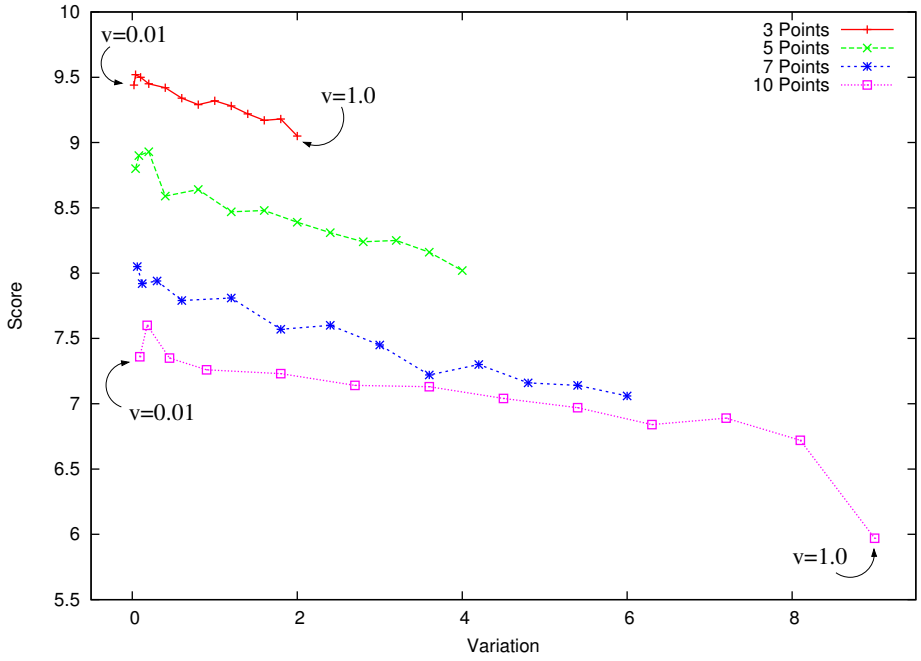


Figure 4.2: Performance of NEAT on the N-Points problem for $N = 3, 5, 7, 10$ and $v = 0.01 \ldots 1.0$. As v increases, the amount of variation required to solve the problem also increases, and NEAT's performance falls. The same trend occurs as N increases. This result highlights NEAT's difficulty in producing solutions with high variation.

that variation in some sense encompasses inflection, since variation will increase as inflection increases, all other things being equal.

This experiment is a simple demonstration of the effect of fracture on NEAT. As the amount of variation required to solve a problem increases, the performance of NEAT falls rapidly. This result supports the hypothesis that NEAT has difficulty in generating high-variation networks. The next experiment examines how the introduction of continuous outputs affects NEAT's ability to generate variation in networks.

4.1.2 Function approximation

The general function approximation problem requires the learning algorithm to evolve neural networks to approximate fixed 1-d functions. Each evolving network is evaluated on a series of numbers representing the input to the function. The network state is cleared before each new input is presented, then the input is fed into the network for $\kappa = 10$ activations. The squared error between the output of the network and the target function is recorded for a series of $\tau = 100$ input points. After the network has been evaluated on all τ input points for a function, the mean squared error is inverted and used as a fitness signal.

This type of function approximation is easy to visualize. Furthermore, it is straightforward to calculate the variation of the optimal solution in this problem. Since a solution is simply a 1-d function, its variation can be calculated as the sum of the absolute value of the function at all τ input points (similar to the process shown in Figure 3.2 and Equation 3.4).

The first group of functions to be approximated follow the form $sin(\alpha x)$. The eight different versions of this sine function (shown in Figure 4.3) have increasing variation (as indicated by the term var), corresponding to larger values of α.

Figure 4.4 shows the performance of NEAT and a linear baseline algorithm on

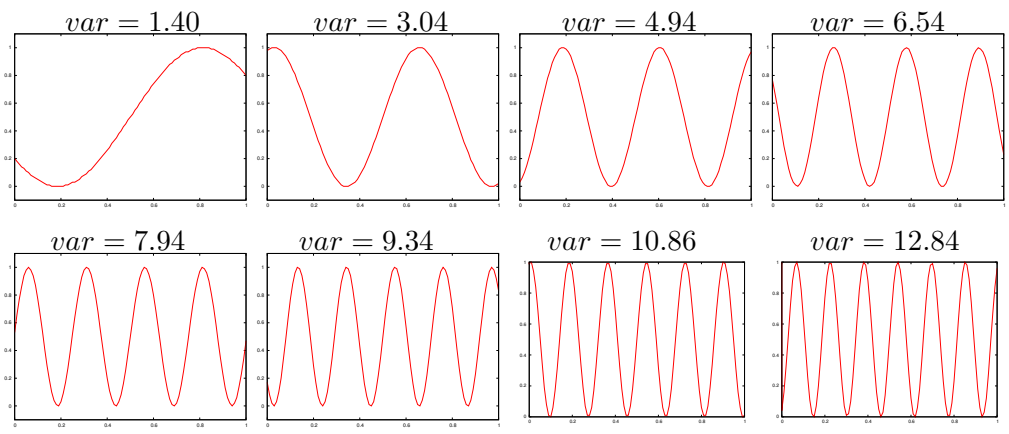


Figure 4.3: Eight versions of a sine wave function approximation problem. Sine waves with higher frequency have higher variation.

these eight sine function problems, averaged over 100 runs. The baseline algorithm is the normal NEAT algorithm without any structural mutation operators, and is therefore limited to evolving a single layer of weights with no hidden nodes. This linear combination of input features is the same initial network topology that NEAT starts with, and is included to provide a sense of scale to these graphs. The horizontal position of each pair of points indicates the variation of the optimal solution for that problem. As variation increases, the score for NEAT drops, confirming the hypothesis that variation is an indicator of problem difficulty for NEAT.

The second set of function approximation experiments involves a much more varied collection of functions. Eighteen different 1-d functions containing differing amounts of variation were chosen arbitrarily, and evaluated in a manner similar to that described in the previous chapter. As before, the minimal amount of variation required for an optimal solution was explicitly calculated using Equation 3.4. These functions are shown in Figure 4.5.

Figure 4.6 shows the average score for NEAT and the linear baseline algorithm on each of the 18 functions. As before, the horizontal axis indicates the variation of the function, and the vertical axis shows the average score achieved by

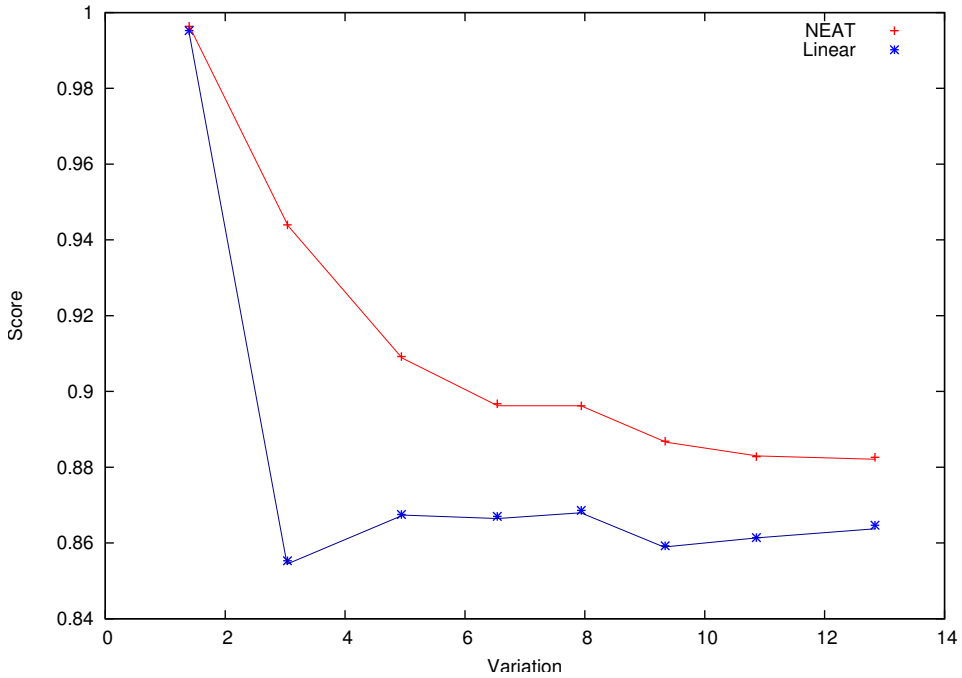


Figure 4.4: NEAT's performance for the eight sine wave function approximation problems from Figure 4.3. Although NEAT can find solutions to low-variation problems, its performance falls quickly as the amount of variation required to solve the problem increases.

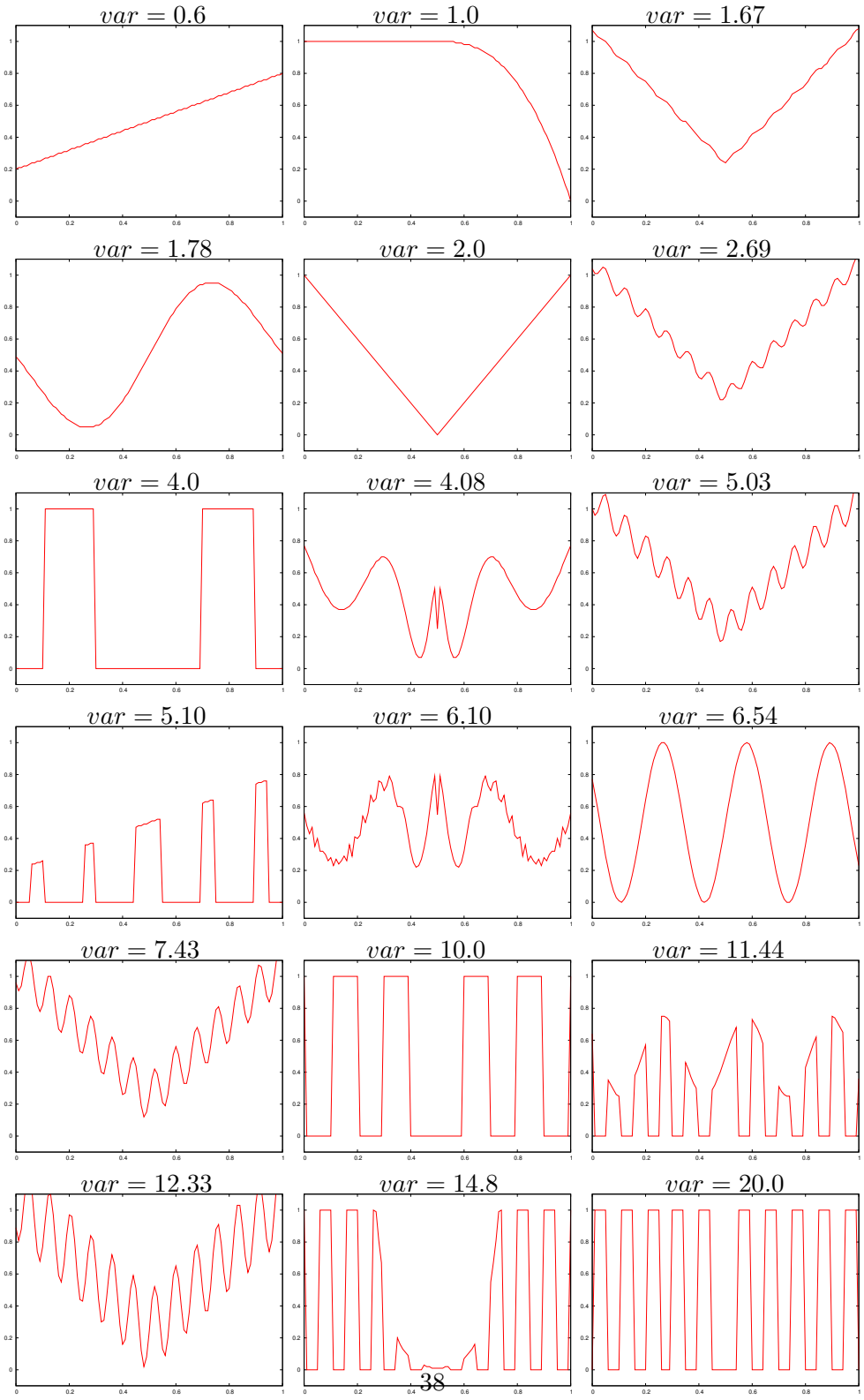


Figure 4.5: The 18 various functions used in the second set of function approximation experiments. The minimal variation required to approximate each of these functions is easily computed using Equation 3.4.

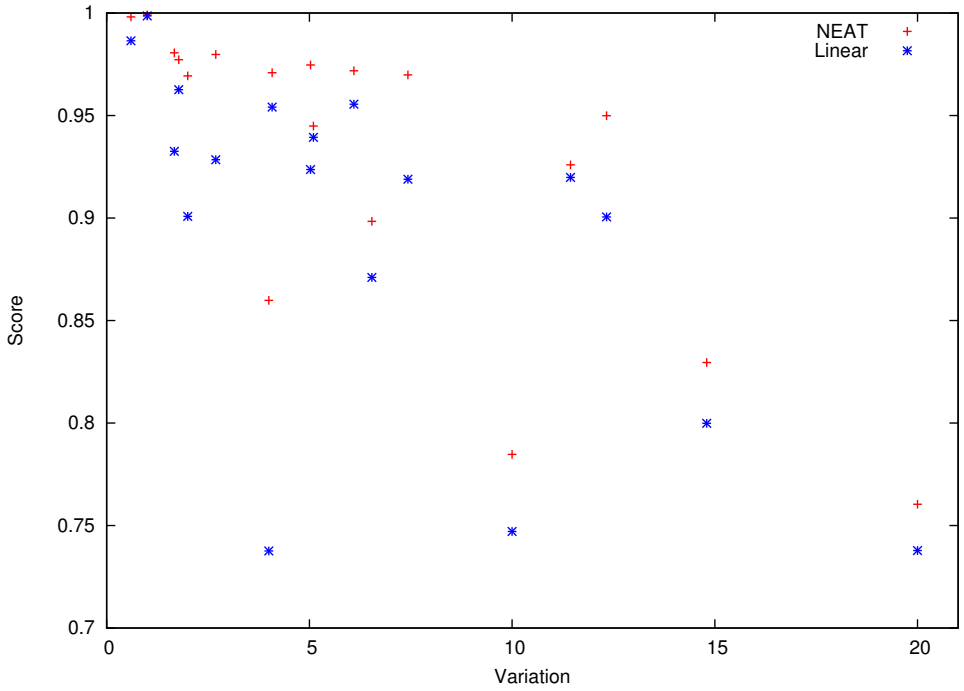


Figure 4.6: Performance of the NEAT algorithm on each of the 18 functions from the second set of function approximation experiments, shown in Figure 4.5. In general, NEAT's performance falls as the variation of the optimal policy for a problem increases.

NEAT. Although the trend is not as obvious as it is for the sine wave functions, there is still a clear relationship (and a correlation coefficient of -0.77) between the variation of the optimal policy for each problem and the performance of NEAT on that problem.

4.1.3 Concentric spirals

The concentric spirals problem is a classic supervised learning benchmark task popularized by the Cascade Correlation literature. Originally proposed by Alexis Wieland (Potter and Jong, 2000), the problem consists of identifying points from two intertwined spirals. Solving the concentric spirals problem involves repeatedly tagging nearby regions of the input space with different labels, which intuitively

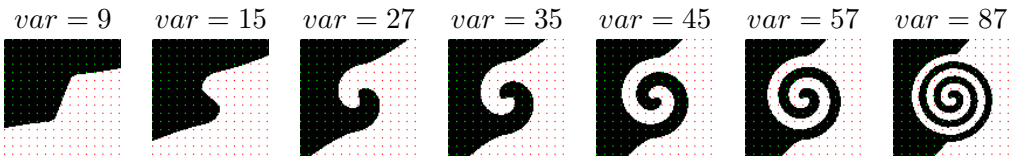


Figure 4.7: Seven versions of the concentric spirals problem that vary in the degree to which the two spirals are intertwined. The colored dots indicate the discretization used to generate data from each spiral. As the spirals become increasingly twisted, the variation of the optimal policy increases.

matches the description of a fractured problem.

In order to examine the effect of changing amounts of fracture on NEAT, seven different versions of the problem were created. These versions of the spirals problem are shown in Figure 4.7. Each of these problems varies in the degree to which the two spirals are intertwined. As the spirals become increasingly intertwined, the variation of the optimal policy (which classifies a set of points as being on one spiral or the other) increases.

Figure 4.8 shows the average score for NEAT and the linear baseline algorithm averaged over 25 runs. Again, scores go down as variation increases, showing that the variation of the optimal policy for each problem correlates closely with problem difficulty.

4.1.4 Multiplexer

A challenging benchmark problem from the evolutionary computation community is the multiplexer problem, where an agent must learn to split the input into address and data fields, then decode the address and use it to select a specific piece of data. For example, the agent might receive as input six bits of information, where the first two bits denote an address and the remaining four bits represent the data field. The two address bits indicate which one of the four data bits should be set as output.

This section describes experiments with four versions of the multiplexer prob-

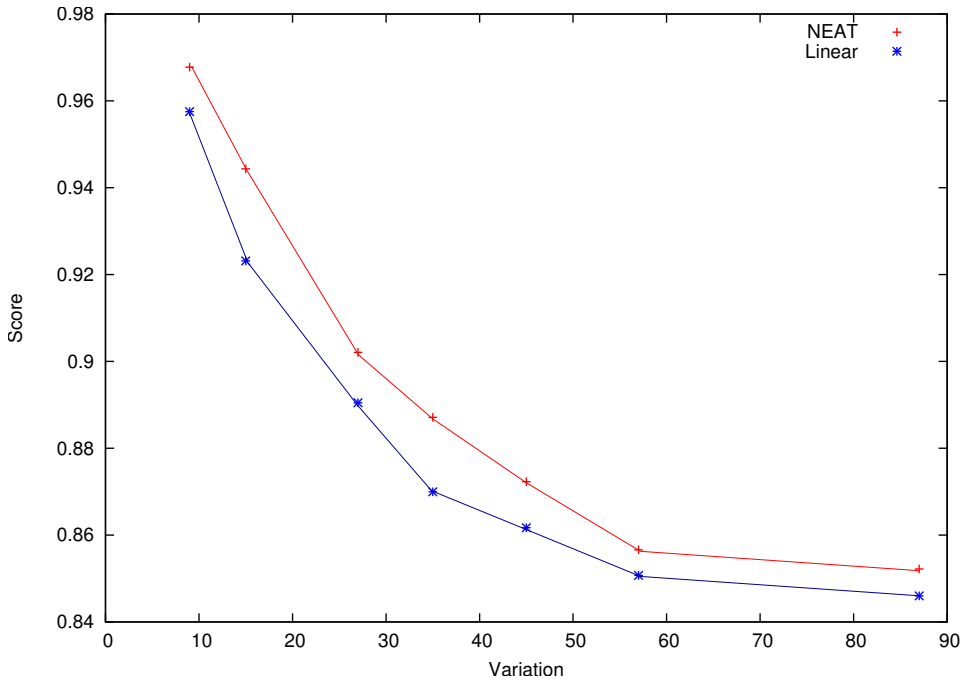


Figure 4.8: Average score for NEAT and the linear baseline algorithm on seven versions of the concentric spirals problem. NEAT's performance falls as the amount of variation required to solve the problem increases.

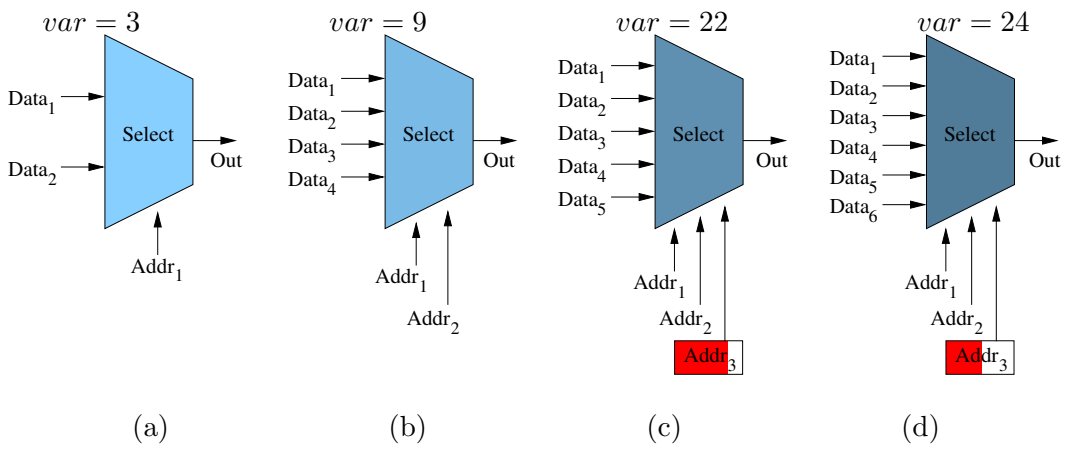


Figure 4.9: Four versions of the multiplexer problem, where the goal is to use address bits to select a particular data bit. For (c) and (d), not all of the values for the third address bit were used. The amount of variation required to solve the multiplexer problem increases as the number of total inputs (address bits plus data bits) increases.

lem, which are shown in Figure 4.9. These four problems differ in the size of the input, ranging from three inputs (one address bit and two data bits) to nine inputs (three address bits and six data bits). Note that in order to make the problem tractable, not all inputs involving the third address bit are used for the two largest versions of the problem.

Computing the minimum variation required to solve the multiplexer problems can be accomplished in a manner similar to the function approximation case. An optimal policy that maps each input to an output is simply a function that is restricted to outputs of 0 or 1. The variation of this function over the valid inputs that are presented during training (the different binary combinations of address and data bits) can be calculated using Equation 3.3. As the number of inputs increases, the variation of the optimal solution also increases. This examination of multiple versions of the problem allows the impact of variation on performance to be measured.

Each version of the multiplexer problem effectively defines a binary function

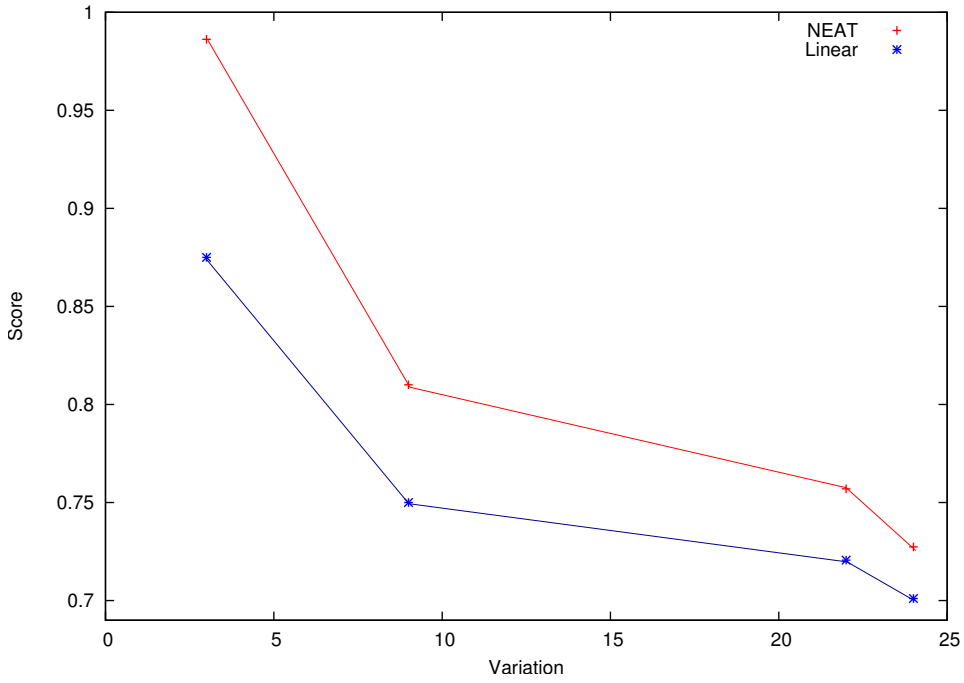


Figure 4.10: Performance of NEAT and the linear baseline algorithm on four versions of the multiplexer problem. As the number of inputs increases, the performance of NEAT declines, again demonstrating the difficulty with increasing variation.

from the input bits to a single output bit. During learning, every possible combination of inputs (given the constraints on address and data bits) was presented to each network in turn. As before, network state was cleared between consecutive inputs. The fitness for each network was the inverted mean squared error over all inputs.

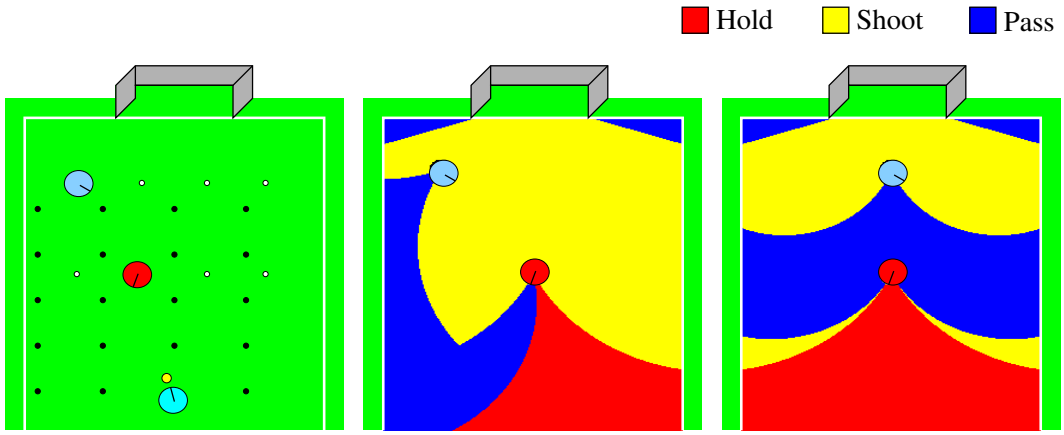
Figure 4.10 shows the performance of NEAT and the linear baseline algorithm on these four versions of the multiplexer problem. The first thing to note is that as the number of inputs increases, the required variation of an optimal solution (plotted on the horizontal axis) increases. While NEAT is able to perform quite well on the simplest version of the problem, its performance falls off quickly as the required variation increases.

4.1.5 3-Player soccer

The 3-player soccer problem represents a class of high-level strategy problems that have previously proved difficult for NEAT to learn. Instead of directly controlling the low-level actions of an agent, a successful network must analyze the state of a soccer game and choose between several predefined "macro" behaviors. The correct behavior changes repeatedly as the network encounters different states, giving this problem a fractured quality. Perhaps the most important feature of the 3-player soccer problem is that it represents an interesting class of problems: It is a decision task where a number of subroutines need to be integrated to achieve a cohesive high-level behavior.

Figure 4.11a shows a typical setup for the 3-player soccer problem. The input to a network consists of the player's position, the position of a single opponent, and the position of a teammate. The player starts with the ball, and must choose between holding the ball, passing to the teammate, or shooting on the goal. Eight versions of this soccer problem were created, each characterized by a different level of discretization. The simplest problem was discretized into a total of 31 states, whereas the most complicated version discretized into 321 states. This discretization was accomplished by varying the position of the player in two dimensions, and varying the teammate and opponent positions along different horizontal lines. The black and white dots in Figure 4.11 show the discretization for the 321-state version of the problem.

As with the multiplexer problem described in Chapter 4.1.4, the minimal variation required for an optimal solution was computed by treating the optimal policy as a function and then applying Equation 3.3. In this case, the optimal function mapped the three state inputs to three binary outputs. Increasing the discretization led to more comprehensive evaluations of an agent's skill. The variation of optimal policies for these eight versions of the 3-player soccer problem ranged



(a) (b) (c)

Figure 4.11: (a) A typical example of the 3-player soccer problem. The player (shown at the bottom-center near the ball) must choose one of three high-level behaviors: hold, pass, or shoot on goal. The black and white dots indicate possible positions for the agents for a 321-state version of the problem. (b)-(c) Optimal actions for different player locations given two possible configurations of teammate and opponent. As the player moves, the optimal action changes frequently and abruptly. By increasing the discretization of the input space, versions of the problem requiring increasing amounts of variation can be created. Thus 3-player soccer is an example of an interesting high-level learning problem that possesses a tunable and fractured decision space.

from $var = 4.33$ for the simplest problem to $var = 101.0$ for the most discretized and difficult version.

Figure 4.12 shows the performance of NEAT and the linear baseline algorithm on the eight versions of the 3-player soccer problem. As the number of states increases, the variation of the optimal policy for each problem also tends to increase. The performance of NEAT falls as it is required to generate increasingly varied solutions.

The 3-Player Soccer problem provides an example of an interesting high-level learning problem that possesses a fractured decision space. Figures 4.11b and 4.11c illustrate the nature of this fracture by showing how the correct action changes

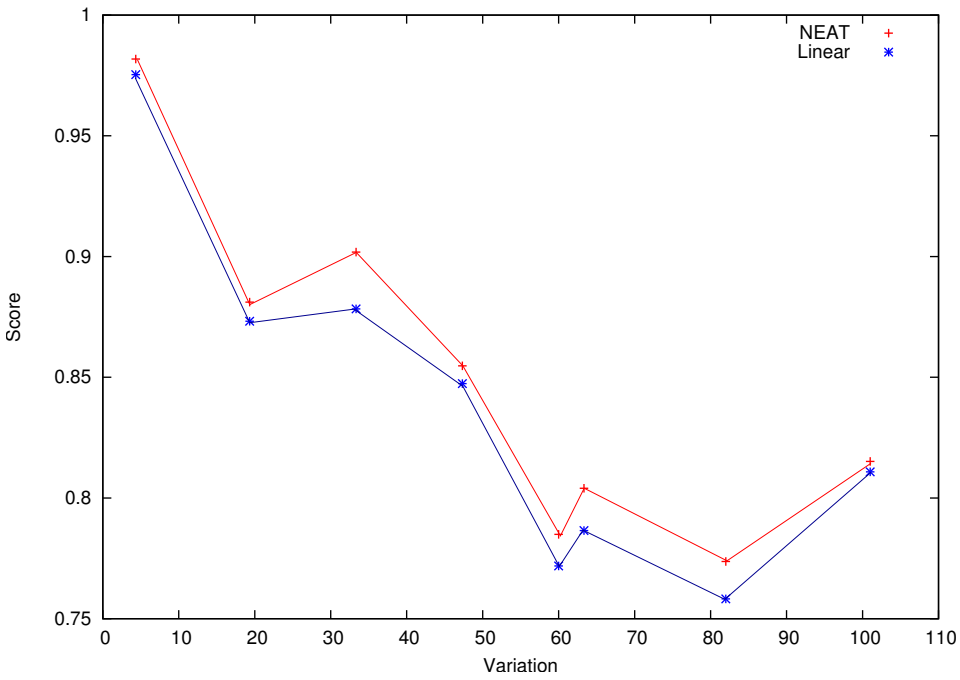


Figure 4.12: Performance of NEAT and the linear baseline algorithm for eight versions of the 3-player soccer problem. NEAT's performance falls as the variation of the optimal policy for each problem increases.

depending on the player's position. The colors on the field — which show the optimal action for the player if it were at that location — can vary quite abruptly in certain places. For example, in Figure 4.11c, the best action when blocked by the opponent is to hold the ball. As the player moves around the opponent, it becomes possible to pass the ball to the teammate, who has a clear shot on the goal. However, there is a thin area where the opponent would be able to intercept a pass to the teammate, which makes the optimal action for that area to take a shot on the goal. Such rapid transitions between optimal actions are the hallmark of a fractured problem.

4.1.6 Conclusion

The experiments shown in Chapters 4.1.1 through 4.1.5 evaluated NEAT on a variety of reinforcement learning problems. Each of these problems was created with a tunable parameter allowing different versions of the problem to be created that required different amounts of variation to solve. The resulting performance of NEAT showed a clear correlation with problem variation: as the amount of variation required to generate an optimal solution increases, the performance of NEAT decreases. This result supports the hypothesis that NEAT has difficulty with fractured problems.

4.2 Measuring Evolved Variation

One method of gaining insight into the results of the previous sections is to measure how much variation there actually is in the solutions that NEAT produces. Figure 4.13 shows this data for several of the problems described above. Each point in the figure represents the average of either 25 or 100 runs of NEAT on a particular problem. The horizontal axis measures the difference between the average variation in NEAT's solutions and the variation of the optimal policy; the larger the difference, the further NEAT was from producing a network with enough variation to

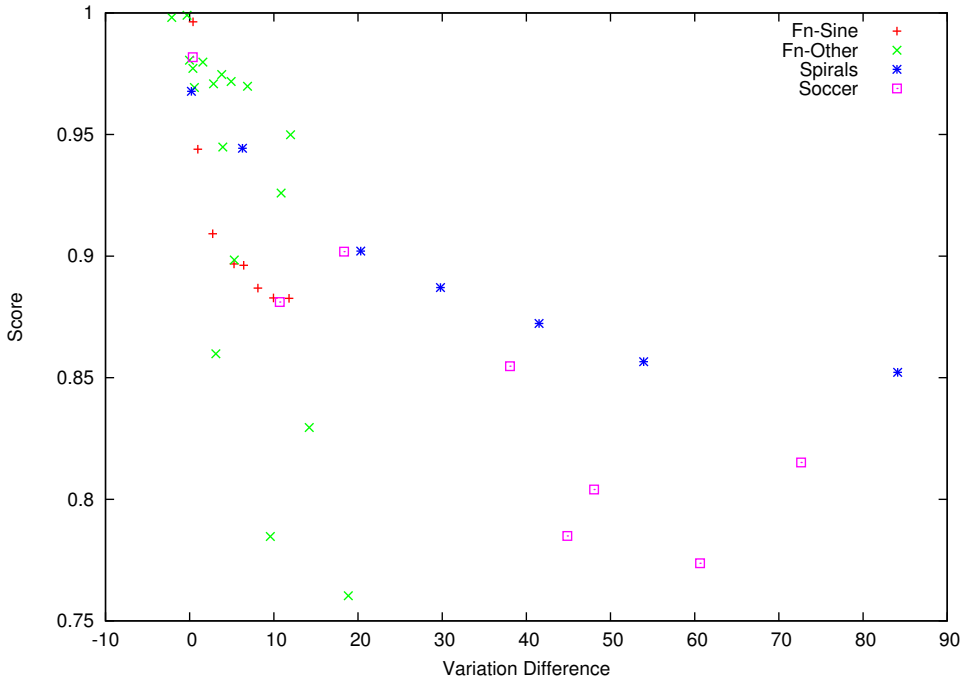


Figure 4.13: Difference in variation between optimal policies and policies evolved by NEAT, and the corresponding effect on score. When NEAT cannot produce enough variation to match the optimal level of variation required for a problem, its performance suffers.

solve the problem. The vertical axis shows performance.

Figure 4.13 shows a clear relationship between variation and score. As the gap in variation between optimal and what NEAT can produce increases, performance goes down. This result provides evidence in support of the hypothesis that NEAT has difficulty in certain problems because it cannot generate solutions with enough variation. The next section aims at understanding why this happens.

4.3 Analysis of NEAT

The results above suggest the NEAT has difficulty producing neural networks with high variation. In order to better understand this deficiency of the algorithm, a series

of experiments were performed that examined the networks produced by typical runs of NEAT on a fractured problem.

NEAT was run for 100 trials on the fifth sine-wave function approximation task (described above in Chapter 4.1.2), which represents a fairly fractured yet easy-to-analyze problem. Each run produced a single network that yielded a certain score. Interestingly enough, there was significant variation in the quality of these champion networks: Some networks performed quite well, while others failed to solve the task completely. This result is illustrated in Figure 4.14, which shows the significant difference ($p > 0.99$) in variation between the best 10 and worst 10 runs. The analysis below aims to understand this difference in performance between successful and unsuccessful runs of NEAT.

4.3.1 Network size

After examining the differences between the successful and unsuccessful runs of NEAT, it became clear that when NEAT worked, it produced networks that were significantly larger than those produced when it failed. Figure 4.15 compares the number of nodes and connections in the successful and unsuccessful groups of champion networks. Both node and connection counts are higher in the best 10 runs.

Another way of exploring the role of network size in fractured problems is to estimate the maximum amount of variation that networks of various sizes can produce. To test this, a series of 100 experiments were run where 2-input, 1-output networks of increasing size were evolved to produce as much variation as possible. The input space for this "maximizing variation" problem was uniformly divided into roughly 200 points. An evaluation consisted of evaluating a network on a each of these points and noting the output that was produced from the single output. The score for a network was simply the total variation of the discretized function that the network represented, calculated in a manner described in Chapter 3.3.

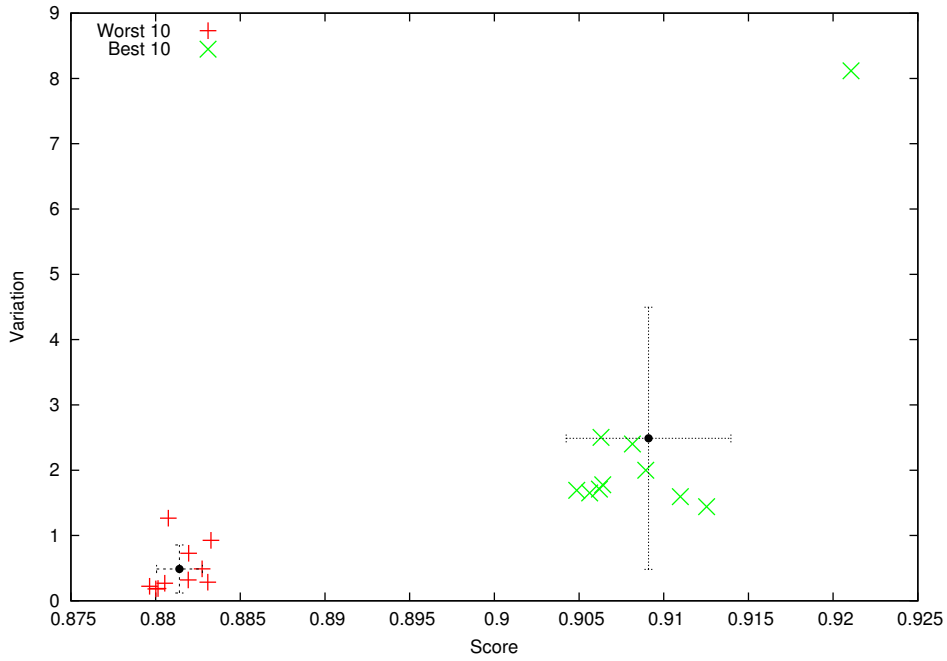


Figure 4.14: The amount of variation that is produced by the best 10 networks and worst 10 networks (selected from a total of 100 networks). Although it frequently fails to do so, some mechanism inside NEAT allows it to occasionally produce networks with high variation. An analysis of these networks can lead to an understanding of why this happens.

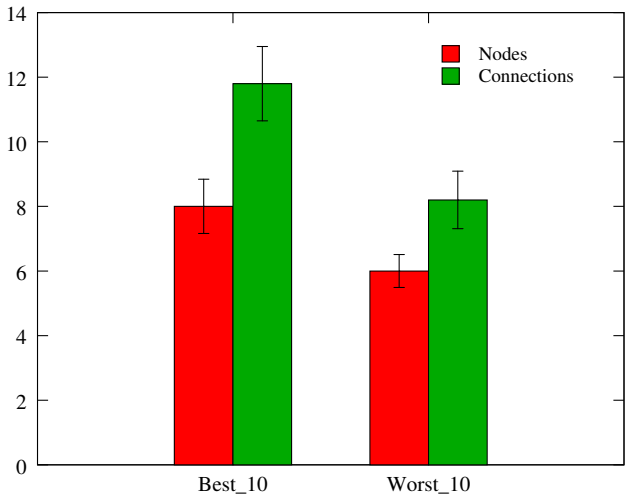


Figure 4.15: A comparison of champion network size between the 10 best and 10 worst runs of NEAT (selected from a total of 100 runs). Successful runs produce champion networks that are significantly larger both in terms of node and connection count, suggesting the importance of being able to build large networks when faced with fractured problems.

The starting point for each run was a single-layer network with no hidden nodes. After determining the amount of variation that this network could produce, either a link or node was added to the network and its weights were randomized. Each run repeated this process 50 times, yielding an estimate of the maximum variation capable of being produced by networks with between zero and 50 mutations. The results, shown in Figure 4.16, show that the amount of variation that small networks can produce is much lower than the level of variation that can be produced by larger networks. If solving a problem requires finding a solution with a high degree of variation, an algorithm that excels at finding small networks might not perform well.

These results suggest that the ability to build large networks is critical in solving fractured problems. The relatively simple and unbiased topological mutations that NEAT uses (add-link and add-node) could very well add structure too

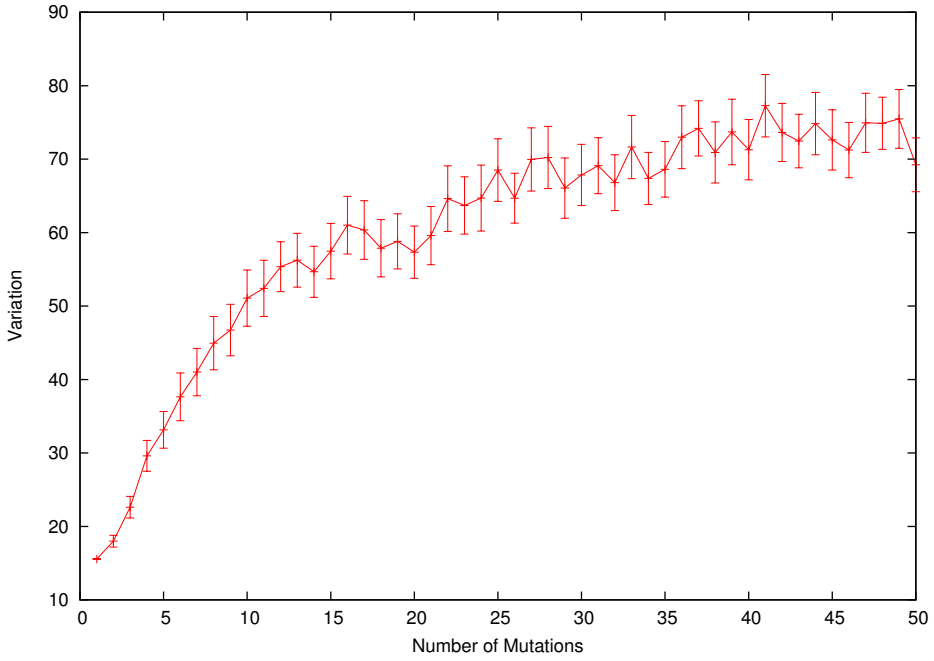


Figure 4.16: The maximum amount of variation produced by networks after a given number of structural mutations. As networks grow in size, their ability to generate large amounts of variation increases. Algorithms that focus on building small networks may not be able to produce networks with large amounts of variation.

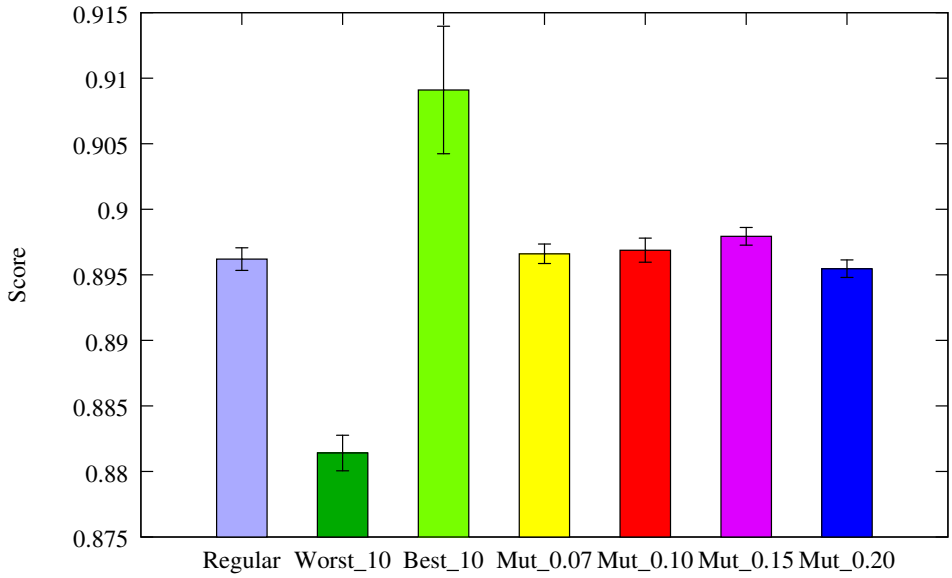


Figure 4.17: The effect of increased mutation rates on learning. Increasing the mutation rate has a negligible or negative effect on performance, suggesting that evolving large networks will need to be done more carefully.

slowly or in a manner that is too ad-hoc to work well on fractured problems. To test the possibility that NEAT was simply adding structure too slowly, four experiments (shown in Figure 4.17) were performed with increased structural mutation rates (increases from $p = 0.05$ to 0.07, 0.10, 0.15, and 0.20). The results of these experiments show that higher mutation rates alone are not enough to improve performance significantly. Instead, it appears that larger networks must be more carefully constructed.

4.3.2 Ablation and lineage analysis

What kind of functionality do the "successful large networks" possess? In order to perform well in a fractured problem, a learning algorithm must be able to generate representations that capture local features of the problem. For example, after the algorithm experiences a new state in the environment, it needs to associate a

specific action for that state. If the problem is fractured, it may not be useful to generalize from the actions of nearby states. Furthermore, any large-scale changes the algorithm may attempt to make could disrupt the individual actions tailored for other states. Therefore, the algorithm must be able to make local changes to isolate that particular state from its neighbors and associate the correct action with it.

One possible explanation for the rareness and efficacy of these large networks is that producing highly-varied solutions to a problem requires the manipulation of *local decision regions* which are difficult to construct in an ad-hoc manner. If a learning algorithm is able to make repeated local changes to a solution by creating these local decision regions, then an accumulation of such changes could result in a solution with high variation. On the other hand, if an algorithm cannot assemble enough network structure to represent such local decision regions, it could have difficulty solving fractured problems. This hypothesis would explain the results presented above. In addition, it also suggests that the difference between the best and worst runs shown in Figure 4.14 is related to the creation of local structures.

In order to test this hypothesis, the lineage of the ten best and worst runs was analyzed. The complete sequence of all network topologies explored by the algorithm for each run was examined, from the initial population to the final population. Each network in each generation that managed to improve in fitness over the fitter of its two parents was noted. The characteristics of the output of these parent-child network pairs was then examined, with the goal of identifying local changes between successful parent-child pairs.

The notion of a local change from parent to child was defined in the following manner. Since these networks were evolved to approximate a function with one input and one output, the output from each parent-child network pair was thought of as two discretized 1-d functions, f_{parent} and f_{child}. The difference f_{diff} between f_{parent} and f_{child} was computed at each point of the discretization. The values in f_{diff}

were then summed together (starting with the largest values and working down to smaller values) until the sum exceeded $P_{diff} = 50\%$ of the total area under f_{diff}. The number of chunks of f_{diff} required to reach this sum (labeled ρ, measured as a fraction of the total number of chunks that f_{diff} was discretized into) represents the amount of the input responsible for $P_{diff}\%$ of the area under the function. When the difference between f_{parent} and f_{child} is localized to a small area, ρ is small. As the difference between the two functions becomes less localized (e.g. when the functions are separated by a constant factor) ρ will increase. This difference therefore serves as a metric that describes the locality of the change between each parent-child network pair.

Figure 4.18 illustrates this locality computation. Suppose a given network has two children that result in an increase in score. One child represents a local change in functionality from its parent, whereas the other child makes global changes. Performing the computation above on the sorted differences between the parent and its two children reveals that ρ is much smaller for the network with local changes.

Figure 4.19 shows how frequently changes with various amounts of locality occurred in the best 10 and worst 10 runs. The best runs tended to make significantly more changes that were local, whereas the worst runs tended to make more broad changes ($p > 0.95$). This result supports the hypothesis that the ability to make local changes is crucial in solving highly-fractured problems.

To further confirm this hypothesis, the weights of the 10 worst and 10 best networks were systematically ablated 100 times. Each ablation resulted in a pair of networks (similar to the parent-child networks above) that were then measured to determine how local the ablation was. Figure 4.20 compares the average locality of change to the average total change for the best and worst groups of networks. While the total change from the ablation did not differ significantly between the

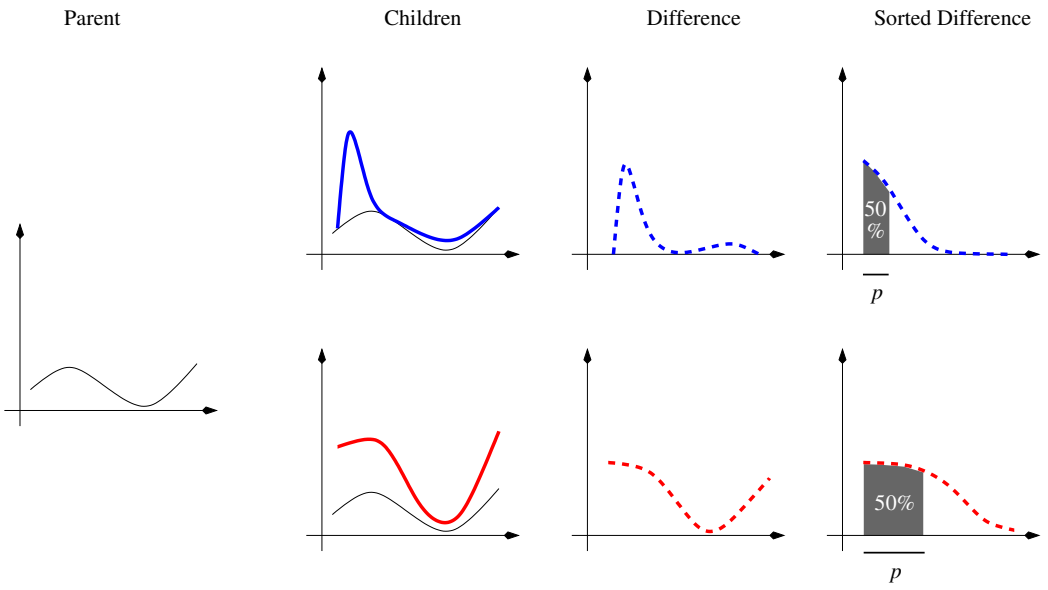


Figure 4.18: An illustration of a locality metric between two networks. The functional output of a parent network is compared to the output of two children networks. By examining the sorted difference in output between each parent-child pair, it becomes clear that the child with more local modifications to its functionality has a much smaller value of ρ.

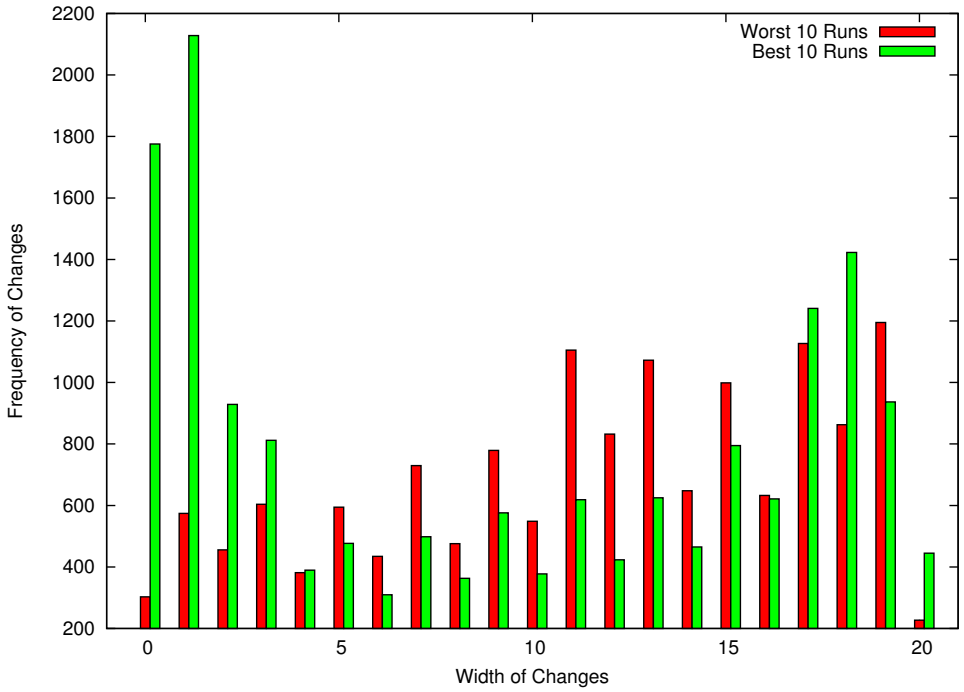


Figure 4.19: A comparison of the locality of changes made by the best 10 and worst 10 runs of NEAT (selected from a total of 100 runs). When NEAT does well, it tends to make changes to networks that have a highly local effect on network output. This result suggests that locality is important in solving fractured problems.

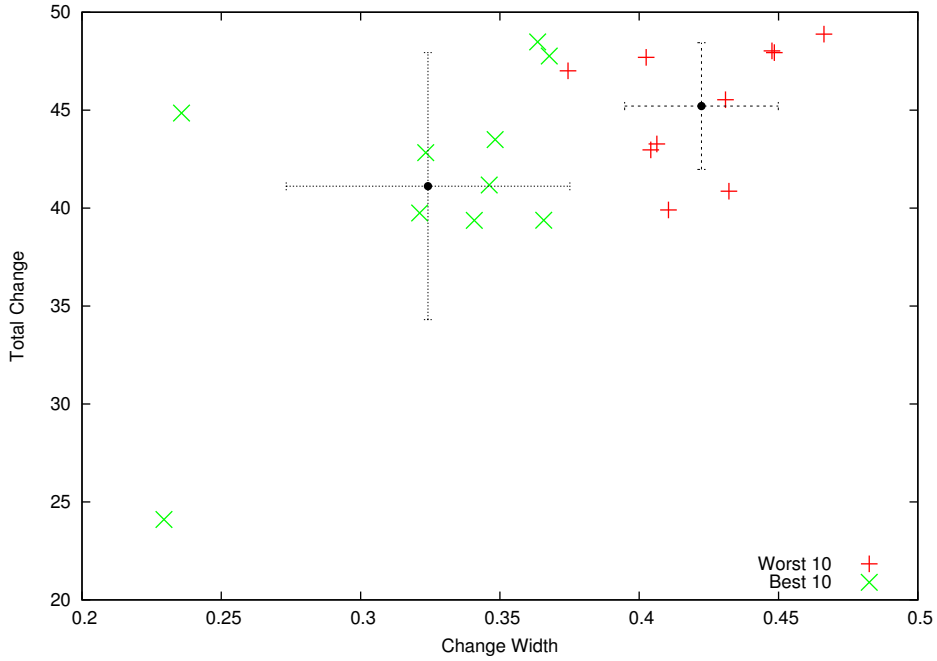


Figure 4.20: An ablation study comparing the best 10 and worst 10 runs of NEAT (selected from a total of 100 runs). After ablation, although the total amount of change remains the same, the best networks tend to change more locally than do the worst networks.

best and worst groups, the best networks did tend to produce more local changes when ablated. This result offers more support to the hypothesis that successful runs of NEAT tend to produce networks that are capable of being changed locally.

4.4 Conclusion

This chapter described experiments on domains with solutions that contained varying degrees of fracture. On problems where solutions required a high level of fracture, NEAT performed poorly, suggesting that fracture is a useful measurement of problem difficulty and that NEAT has difficulty assembling neural networks that exhibit large amounts of variation.

Further experiments showed that NEAT can occasionally produce reasonable solutions to fractured problems. A closer examination of this result revealed that NEAT performs well when it is able to generate large networks that model local decision regions. This observation suggests an opportunity for designing stronger neuroevolution methods by systematically utilizing locality. The next chapter examines related work on local learning and proposes how neuroevolution algorithms might be modified to take advantage of this idea.

Chapter 5

Solution: Modeling Fracture with Locality

In the previous chapter, NEAT was shown to have difficulty in generating networks with enough variation to solve highly-fractured problems. Several experiments showed that NEAT has difficulty generating local decision regions. The concept of localized change – as opposed to large-scale, global change — has appeared before in many parts of the machine learning community. This chapter reviews the related literature for previous approaches to generating local decision regions and solving fractured problems, and proposes two methods of incorporating these concepts into neuroevolution algorithms.

In order to perform well in a fractured problem, a learning algorithm must be able to generate representations that capture local features of the problem. For example, after the algorithm experiences a new state in the environment, it needs to associate a specific action for that state. If the problem is fractured, it may not be useful to generalize from the actions of nearby states. Furthermore, any large-scale changes the algorithm may attempt to make could disrupt the individual actions tailored for other states. Therefore, the algorithm must be able to make

local changes to isolate that particular state from its neighbors and associate the correct action with it.

5.1 Supervised Learning

One promising method for learning local features is radial basis function (RBF) networks (Gutmann, 2001, Moody and Darken, 1989, Park and Sandberg, 1991, Platt, 1991). RBF networks originated in the supervised learning community, and are usually described as neural networks with a single hidden layer of basis-function nodes. Each of these nodes computes a function (usually a Gaussian) of the inputs, and the output of the network is a linear combination of all of the basis nodes. Training of RBF networks usually occurs in two stages: The locations and sizes of the basis functions are determined, and then the parameters that combine the basis functions are computed. Figure 5.1 shows a simple example of how an RBF network can isolate local areas of the input space with fewer mutable parameters than a sigmoid-node neural network. For an overview of RBF networks in the supervised learning literature, see (Ghosh and Nag, 2001).

The local processing in RBF networks has proved to be useful in many problems, frequently competitive with or outperforming other function approximation techniques (Lawrence et al., 1996, Wedge et al., 2005). Such local approaches have been particularly useful on supervised learning problems that might be considered fractured, like the concentric spirals classification task (Chaiyaratana and Zalzala, 1998). This success suggests that an RBF approach could be useful for fractured reinforcement learning problems as well. Of course, supervised RBF algorithms frequently take advantage of labeled training data when deciding how to build and tune RBF nodes, and such data is not available in reinforcement learning. Furthermore, most of the network architectures proposed in supervised RBF algorithms are fixed before learning or are constrained to be highly regular (e.g. a single hidden layer of

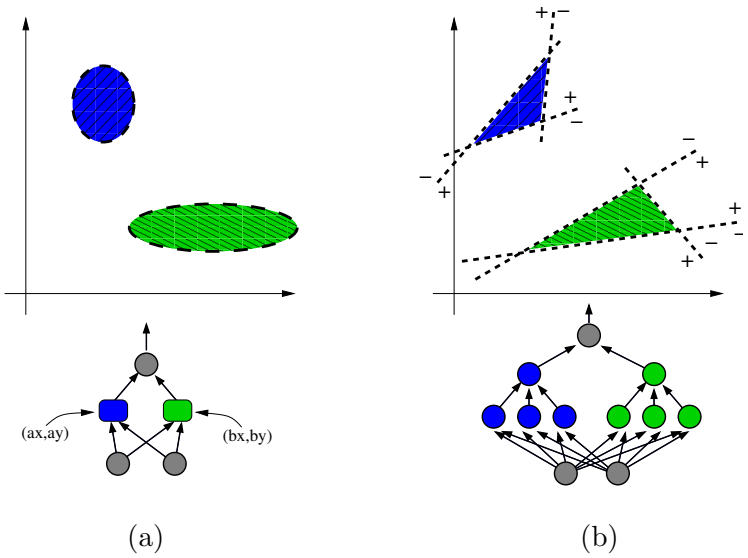


Figure 5.1: A comparison of how an RBF network (a) and a sigmoid-node network (b) might isolate two specific areas of a 2-d input space. The local functionality of the RBF network can identify comparable spaces using far fewer parameters.

RBF nodes). This constraint could limit the ability of the learning algorithms to find an appropriate representation for the problem at hand.

The Cascade Correlation algorithm is a method of building neural networks that is related to RBF networks. Introduced by Fahlman and Lebiere, Cascade Correlation works by incrementally growing a network with repeated training iterations (Fahlman and Lebiere, 1990). The first half of each iteration trains a newly-added hidden node (connected to previous hidden nodes and inputs) to correlate with errors in network output. Next, the hidden node is connected to the network outputs and the remaining weights are trained to use this new source of input to reduce network error. Cascade correlation was shown to be highly effective at forming the twisted decision regions of the concentric spirals problem, but is highly dependent upon supervised input.

Another relevant area in the supervised learning community is "deep learning" (Hinton and Salakhutdinov, 2006). Proponents of deep learning argue that

neural networks with a large number of nodes between input and output are able to form progressively high-level and abstract representations of input features (Bengio, 2007, LeCun and Bengio, 2007). However, training such deep networks with standard techniques like backpropagation can be quite difficult due to the diminished error signal associated with propagation of error over many connections. In order to solve this problem, deep learning networks are pre-trained using unsupervised learning to cluster input patterns into distinct groups. This pre-training sets the weights of the network close to good values, which then allows backpropagation to run successfully.

The arguments for deep learning are complementary to those for constructive neuroevolution; both communities value more complicated network structures that can hold sophisticated representations of input data, as opposed to single-layer architectures. The two approaches diverge in their method of building large networks, of course, as deep learning does not construct networks and the second stage of deep learning relies on supervised feedback. However, it might be possible to incorporate deep learning's initialization of network weights into neuroevolution. It is certainly possible that starting with network weights that are close to optimal could significantly improve performance.

5.2 Reinforcement Learning

In contrast to the approaches described above, reinforcement learning algorithms are designed to solve problems where labeled training data is unavailable. The idea of local processing has also proved to be effective for value-function reinforcement learning algorithms. Such methods frequently benefit from approximating value functions using highly local function approximators like tables, CMACs, or RBF networks (Kretchmar and Anderson, 1997, Li and Duckett, 2005, Li et al., 2006, Peterson and Sun, 1998, Stone et al., 2006, Sutton, 1996, Taylor et al., 2006). For

example, Sutton used CMACs to get positive results on a set of problems that had previously proved difficult to solve using global function approximators (Sutton, 1996). Asada et al. improved the learning performance of a value-function algorithm by grouping local patches of the state space together that shared the same action (Asada et al., 1995). More recently, Stone et al. found that in the benchmark keepaway soccer problem, an RBF-based value-function approximator significantly outperformed a normal neural network value-function approximator (Stone et al., 2006). Such results suggest that local behavioral adjustments could be useful for policy-search reinforcement learning algorithms — like neuroevolution – as well.

However, none of these local approaches to reinforcement learning have attempted to integrate local processing into an unbiased constructive neural network algorithm. In addition, these local processing approaches have focused on value-function reinforcement learning, where the goal of the function approximator is to approximate a value function. Different dynamics may exist in policy-search reinforcement learning, where the goal is to produce a function that directly maps states to actions.

5.3 Evolutionary Computation

Evolutionary approaches to learning using the cascade correlation architecture have proven to be highly effective on certain benchmark problems like the concentric spirals problem (Potter and Jong, 2000, Tulai and Oppacher, 2002). Although the only concept that these approaches borrow from cascade correlation is the cascade network architecture (i.e. the process of training hidden nodes to correlate with pre-existing error is left out), this topology restriction alone seems to be enough to allow good performance on the concentric spirals problem. It would be interesting to see how well this approach works on other fractured problems, as well as how the other features of NEAT (such as speciation) impact a cascade-architecture algorithm. This

idea will be revisited shortly.

Learning classifier systems (LCS) are another interesting family of algorithms that use local processing to solve reinforcement learning problems. LCS approximate functions with a population of classifiers, each of which is responsible for a small part of the input space. A competitive contributory mechanism encourages classifiers to cover as much space as possible, removing redundant classifiers and increasing generalization. A number of LCS algorithms have been developed that vary both how the classifiers cover the input space and how they approximate local functions (Butz, 2005, Butz and Herbort, 2008, Wilson, 2008, 2002, Lanzi et al., 2006, 2005, Bull and O'Hara, 2002, Howard et al., 2008). Of particular interest are approaches like those used in Neural XCSF, which use a fixed-topology or variable-size single-layer neural network to define both conditions and actions of a simple LCS. Although early work examining the role of constructive neural networks in LCS has been promising (Howard et al., 2008), the full potential of a combination of LCS and constructive neuroevolution has not yet been explored.

Several interesting hybrid algorithms have been proposed that use various flavors of genetic algorithms to reduce the amount of required human expertise in supervised learning, usually by automatically determining the number, size, and location of the basis functions (Angeline, 1997, Billings and Zheng, 1995, Chaiyaratana and Zalzala, 1998, Gonzalez et al., 2003, Guillen et al., 2007, 2006, Guo et al., 2003, Maillard and Gueriot, 1997, Sarimveis et al., 2004, Whitehead and Choate, 1996). These approaches still rely on supervised training data, at least in part, and typically are also constrained to produce single-layer network architectures.

For instance, the Global-Local ANN (GL-ANN) architecture proposed by Wedge et al. first trains a single-layer sigmoid-node network, then constructively adds RBF nodes, and finally adjusts all parameters of the network (Wedge et al., 2006). Similarly, the Structural Modular Neural Networks approach uses a genetic

algorithm to evolve single-layer networks with both sigmoid and RBF nodes (Jiang et al., 2003). These approaches are intriguing, in that they combine global approximation with sigmoid nodes with the local adjustments of RBF nodes, but the resulting network architectures are still quite regular when compared to the unbiased architectures that algorithms like NEAT can discover. These approaches also rely on supervised training data.

Another interesting branch of related work comes from the field of genetic programming (GP), where Rosca developed methods to allow GP to decompose problems into useful hierarchies and abstractions (Rosca, 1997). To the extent that the fracture for a given problem is organized in a hierarchical manner, the adaptive representations introduced by Rosca could be used to bias search towards small, repeated motifs. Of course, the notion of reusable modules of code is easier to define for genetic programs than it is for neural networks. In order to take advantage of this work in GP, it would be necessary to better understand how modular neural networks could be developed.

5.4 Incorporating Locality into Neuroevolution

Although local processing has been examined extensively in supervised learning, value-function reinforcement learning, and evolutionary computation, it has not been utilized in unbiased constructive neural network algorithms such as neuroevolution. Incorporating these concepts into neuroevolution algorithms could significantly improve performance on fractured problems, paving the way for general-purpose learning algorithms. Reducing the amount of domain knowledge required in applying learning algorithms to problems could greatly encourage the widespread adoption of these algorithms and lead to novel applications and solutions.

The following sections describe two algorithms that combine the ideas outlined above with the NEAT algorithm and policy-search reinforcement learning.

Both of these approaches are essentially extensions to the standard NEAT algorithm that are designed to improve performance on fractured problems by either biasing or constraining the types of network structure that NEAT explores.

5.4.1 RBF-NEAT

The first algorithm, called RBF-NEAT, extends NEAT by introducing a new topological mutation that adds a radial basis function node to the network. This mutation is an addition to the normal mutation operators used by NEAT, giving it the ability to generate networks that have both sigmoid-based nodes and basis-function nodes. Like NEAT, the algorithm starts with a minimal topology, in this case consisting of a single layer of weights connecting inputs to outputs, and no hidden nodes. In addition to the normal "add link" and "add node" mutations, with probability $\epsilon = 0.05$ an "add RBF node" mutation occurs (Figure 5.2). Each RBF node is activated by an axis-parallel Gaussian with variable center and size. All free parameters of the network, including RBF node parameters and link weights, are determined by a simple genetic algorithm similar to the one in NEAT (Stanley and Miikkulainen, 2002).

RBF-NEAT is designed to evaluate whether local processing nodes can be useful in policy-search reinforcement learning problems. The addition of a RBF node mutation provides a bias towards local-processing structures, but the normal NEAT mutation operators still allow the algorithm to explore the space of arbitrary network topologies.

5.4.2 Cascade-NEAT

A more drastic alternative to biasing the search for networks is to restrict the topology search to a specific set of structures. The second algorithm, called Cascade-NEAT, restricts the search process to topologies that have a cascade architecture.

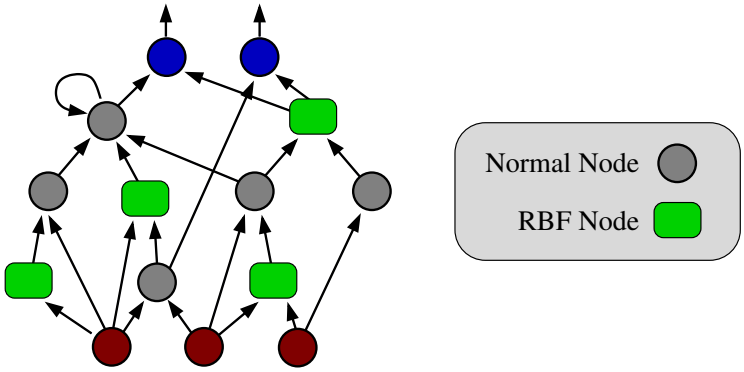


Figure 5.2: An example of the network topology evolved by the RBF-NEAT algorithm. Radial basis function nodes, initially connected to inputs and outputs, are provided as an additional mutation to the algorithm. The simplicity of this algorithm makes it easy to understand how local learning occurs and provides a starting point for creating more sophisticated constructive neural network algorithms.

As described above, the cascade correlation algorithm and its derivatives have proved to be a powerful approach to a certain class of problems. The cascade architecture (shown in Figure 5.3) is a regular form of network where each hidden node is connected to inputs, outputs, and all previously-existing hidden nodes.

Like NEAT, Cascade-NEAT starts from a minimal network consisting of a single layer of connections from inputs to outputs. Instead of the normal NEAT mutations, however, Cascade-NEAT uses an "add cascade node" mutation with probability $\epsilon = 0.05$ that adds a hidden node to the network. This hidden node has inputs from all inputs and existing hidden nodes in the network, and is connected to all outputs. In addition, whenever a hidden node is added, all pre-existing network structure is frozen in place. Thus, at any given time, the only mutable parameters of the network are the connections that involve the most recently-added hidden node.

The constraint that Cascade-NEAT adds to the search for network topologies is considerable, given the wide variety of network structures that the normal NEAT algorithm examines. The idea is that this restriction results in gradual abstraction and refinement, which allows the discovery of solutions with local processing

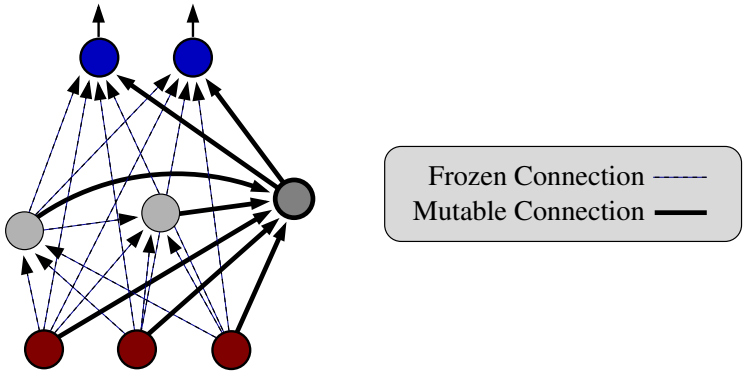


Figure 5.3: An example of a network constructed by Cascade-NEAT. Only connections associated with the most recently added hidden node are evolved. Compared to NEAT and RBF-NEAT, Cascade-NEAT constructs networks with a very regular topology that should result in local processing.

structure.

5.5 Conclusion

The experiments presented in previous chapters showed that NEAT has difficulty in performing well on problems requiring solutions with a large amount of variation. Further examination revealed that NEAT occasionally performs well, but only when it is able to produce solutions that feature local processing structure.

This chapter reviewed previous approaches to local learning and proposed two new neuroevolution algorithms, Cascade-NEAT and RBF-NEAT, designed to explicitly create such local structure. The next chapter examines the effect of the constraint in Cascade-NEAT — and the effect of the bias in RBF-NEAT — on a series of fractured problems.

Chapter 6

Empirical Analysis

The previous chapter introduced two modified versions of NEAT – Cascade-NEAT and RBF-NEAT — that are based on the idea of biasing or constraining network construction to create local decision regions. Such decision regions should in turn give these algorithms a better chance at solving fractured problems. This chapter describes several experiments designed to evaluate this hypothesis, both by comparing the performance of the algorithms on a set of problems and by analyzing the solutions that the algorithms create.

6.1 Evaluation on Fractured Problems

In order to test the hypothesis that biasing and constraining topology search is beneficial in fractured problems, RBF-NEAT and Cascade-NEAT were compared to the standard NEAT algorithm in the fractured problems introduced in Chapter 4.

6.1.1 N-Point classification

The first comparison was run in the most challenging version of the N-Point Classification problem described in Chapter 4.1.1, i.e. the one with 10 points. Data was col-

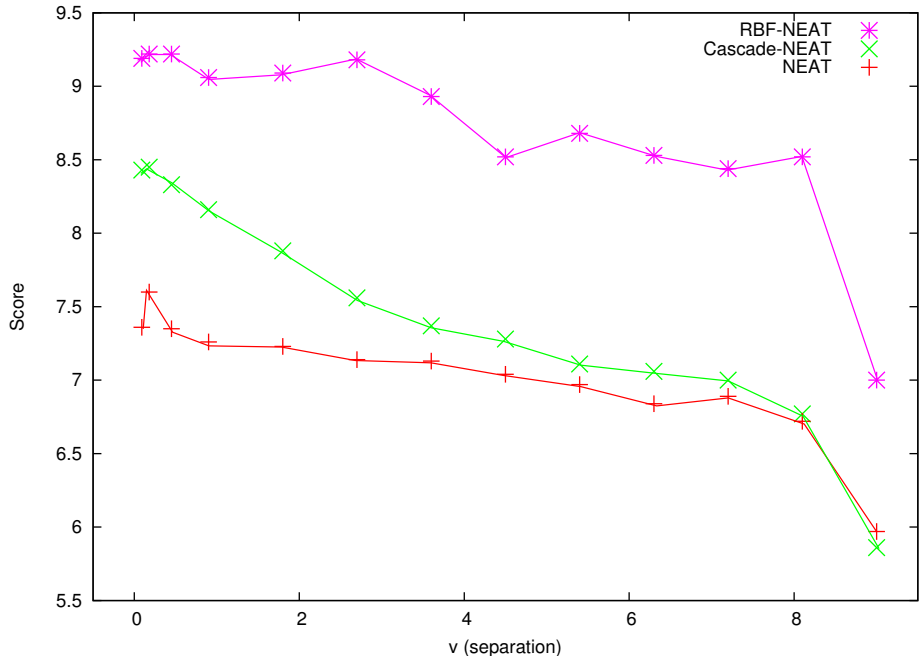


Figure 6.1: A comparison of NEAT, Cascade-NEAT, and RBF-NEAT for 13 versions of the 10-Point Classification problem. RBF-NEAT yields the highest performance across all 13 versions of the problem, and Cascade-NEAT significantly outperforms NEAT on low-variation versions.

lected for 100 runs of each algorithm. As can be seen in Figure 6.1, Cascade-NEAT is significantly better than NEAT when the problem does not require much variation (corresponding to points on the left side of the figure, which have low values of v). Cascade-NEAT's performance becomes indistinguishable from NEAT as the variation required to solve the problem increases. However, RBF-NEAT significantly outperforms both algorithms across all 13 versions of the 10-Point Classification problem.

6.1.2 Function approximation

The next evaluation of Cascade-NEAT and RBF-NEAT revisits the two groups of function approximation experiments described in Chapter 4.1.2. For the eight sine

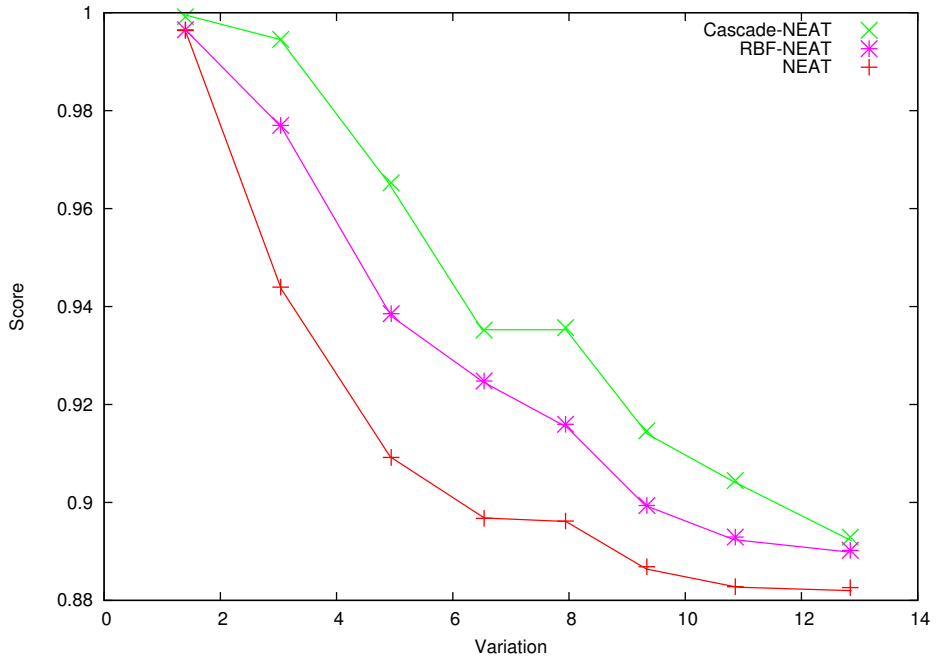


Figure 6.2: Results for the sine wave function approximation problem. Performance drops as variation increases, but RBF-NEAT and Cascade-NEAT are able to outperform the standard NEAT algorithm.

function approximation experiments, NEAT, Cascade-NEAT and RBF-NEAT were evaluated with 100 runs of each algorithm. Figure 6.2 shows the results of these experiments.

In Figure 6.2, each of the eight problems is represented by a vertical cluster of four points. The horizontal position of each cluster indicates the variation of the optimal solution for that problem. Although all algorithms perform similarly for the problem with the least variation, a marked difference appears as the amount of required variation increases. The Cascade-NEAT and RBF-NEAT algorithms generate scores that are significantly higher than the normal NEAT algorithm, supporting the hypothesis that incorporating bias and constraint into network construction can make learning high-variation problems easier.

Figure 6.3 shows the performance of all three learning algorithms on each

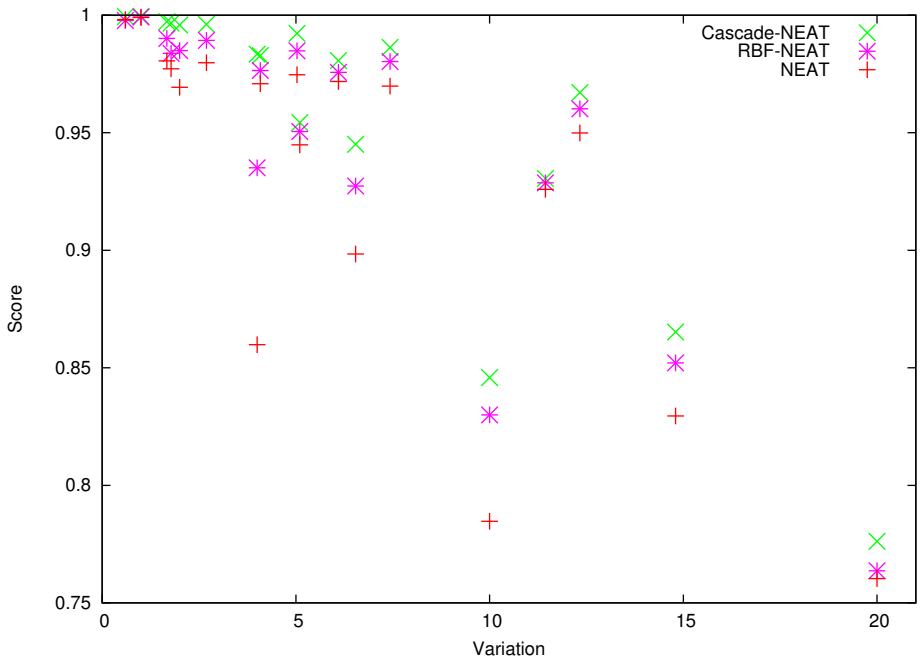


Figure 6.3: Results for the 18 different function approximation problems introduced in Chapter 4.1.2. Both Cascade-NEAT and RBF-NEAT outperform the standard NEAT algorithm significantly in nearly all cases.

of the 18 function approximation problems described in Chapter 4.1.2. As before, Cascade-NEAT and RBF-NEAT are able to make significant improvements over NEAT.

6.1.3 Concentric spirals

The three learning algorithms were also evaluated on the seven concentric spirals problems introduced in Chapter 4.1.3. Figure 6.4 shows the score for each algorithm averaged over 25 runs. Again, scores go down as variation increases, showing that the variation of each problem correlates closely with problem difficulty. However, Cascade-NEAT and RBF-NEAT are able to offer significant increases in performance over that of the standard NEAT algorithm.

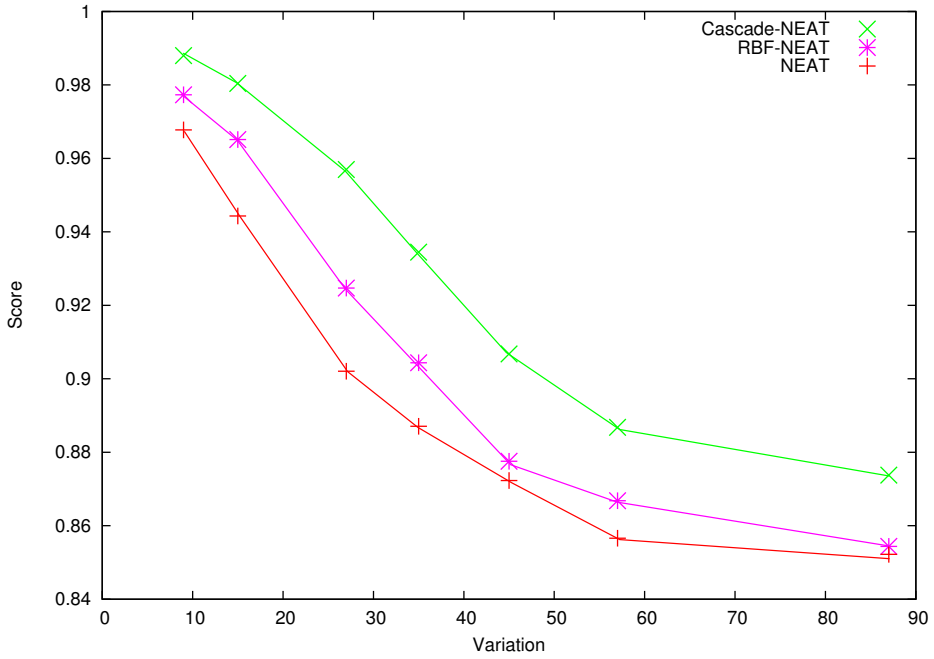


Figure 6.4: Average score for the three learning algorithms on seven versions of the concentric spirals problem. As the variation of the problem increases, performance falls, but Cascade-NEAT and RBF-NEAT are able to outperform the standard NEAT algorithm.

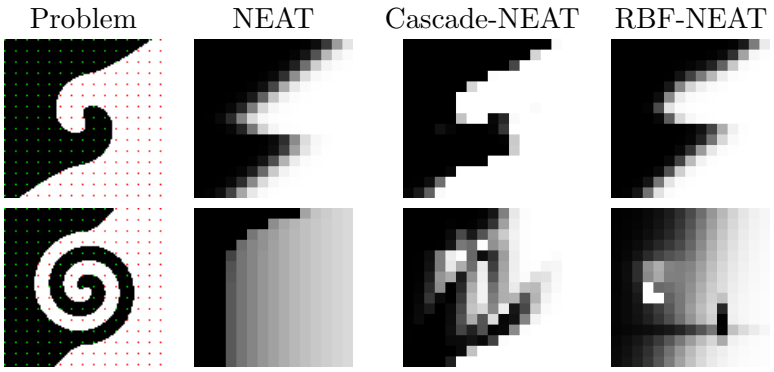


Figure 6.5: Output of the best solutions found by each learning algorithm for two versions of the concentric spirals problem. Cascade-NEAT does a much better job than NEAT at generating the subtle variations required by the more complicated version of the problem.

Figure 6.5 shows the output of the best evolved solutions from each learning algorithm for two of the spiral problems. NEAT is able to find an approximate solution for the simpler problem, but is unable to discover a network that can represent the variation required to do well on the more complex problem. The solutions that Cascade-NEAT generates, while not perfect, clearly are able to encompass more variation than those discovered by NEAT.

6.1.4 Multiplexer

Next, the three learning algorithms were evaluated on four versions of the Multiplexer problem, described in Chapter 4.1.4. Figure 6.6 shows the performance of NEAT, Cascade-NEAT, and RBF-NEAT on these multiplexer problems. As in previous sections, each group of four vertical points represents one of the problems.

While NEAT is able to perform quite well on the simplest version of the multiplexer problem, its performance falls off quickly as the required variation increases. Interestingly, RBF-NEAT does not offer significant increases in performance over regular NEAT for any versions of the problem. However, Cascade-NEAT is able to

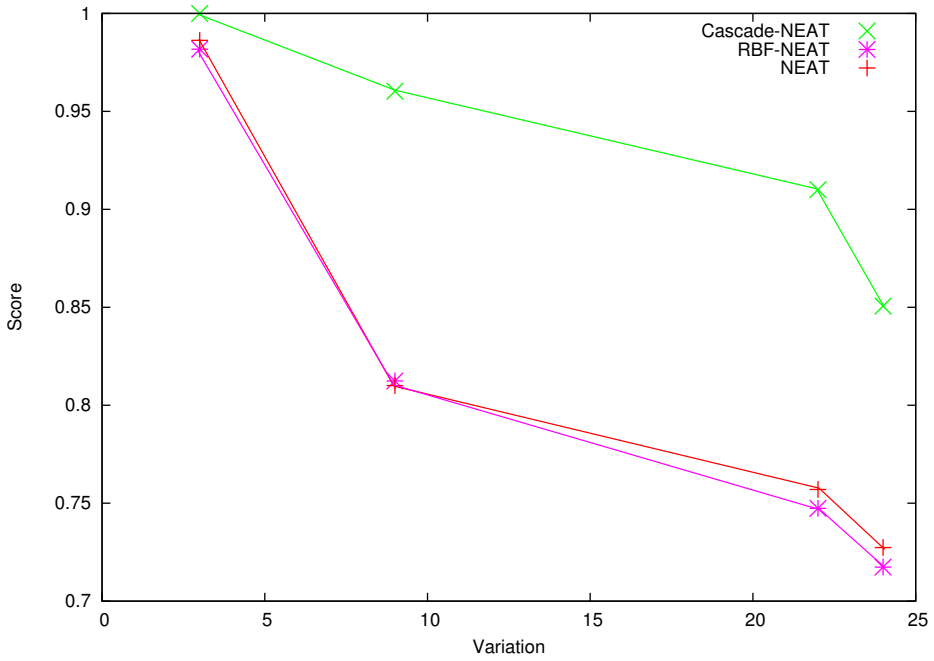


Figure 6.6: Performance of the three learning algorithms on four versions of the multiplexer problem. As the number of inputs increases, so does the require variation of solutions. Cascade-NEAT is able to dramatically improve performance over the other algorithms.

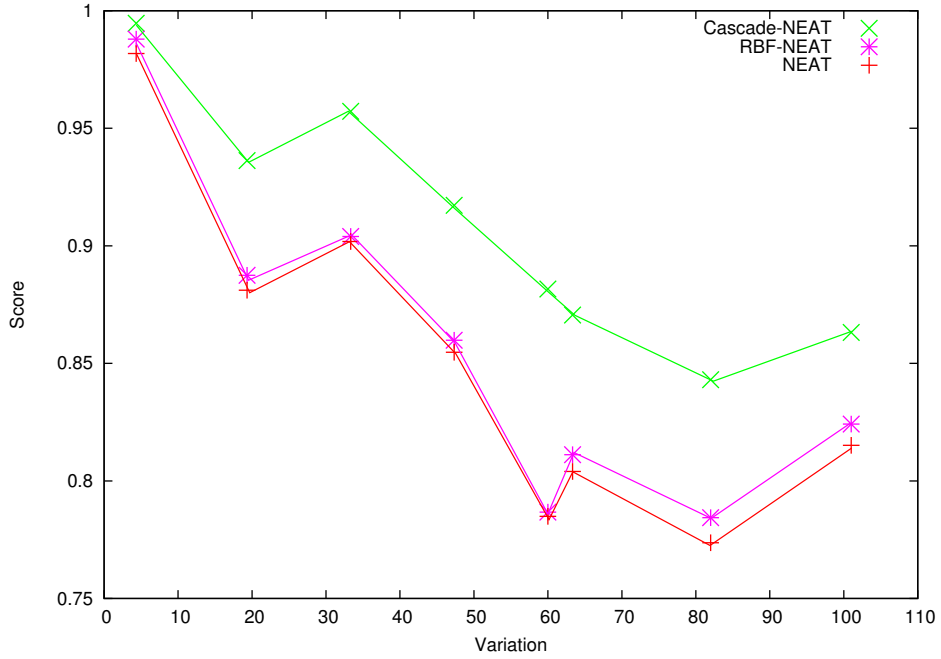


Figure 6.7: Performance of three learning algorithms for the eight versions of the 3-player soccer problem. As the variation of the problem increases, performance tends to fall, but Cascade-NEAT is able to consistently generate higher-quality solutions.

outperform all other algorithms significantly.

6.1.5 3-Player soccer

The final evaluation of RBF-NEAT and Cascade-NEAT revisits the 3-Player Soccer problem, introduced in Chapter 4.1.5. Figure 6.7 shows the performance of the three learning algorithms on the eight versions of the 3-player soccer problem. As before, each problem is represented by a vertical cluster of four points, and the horizontal axis represents the variation of the optimal policy for each problem. These results show that biasing network construction with either RBF nodes or a cascade architecture significantly improves performance.

6.1.6 Conclusion

The results so far offer strong evidence in support of the hypothesis that biasing or constraining network construction can improve performance on fractured problems. Both Cascade-NEAT and RBF-NEAT offer increases in performances over the standard NEAT algorithm, albeit in different situations. Cascade-NEAT seems to work best in general, but when the dimensionality of the input space is low, RBF-NEAT tends to perform best. The next sections examine these results more closely.

6.2 Overfitting vs. Generalization

For problems that are not completely fractured, an ability to generalize from learned concepts is useful. One concern with learning algorithms like Cascade-NEAT and RBF-NEAT is that because they are biased towards discovering local features, they may lack an ability to generalize.

To evaluate this potential problem, a cross-validation study was performed on the most difficult version of one of the more challenging problems, the 321-state 3-Player Soccer problem with $var = 101.0$. Ten versions of the original problem were created, each consisting of a randomly-selected training and testing set of equal size. NEAT, Cascade-NEAT, and RBF-NEAT were trained on each of ten training datasets, and then tested on the respective testing dataset. This division of data into training and testing sets was designed to test the ability of the four learning algorithms to generalize to data on which they were not trained.

The results of this cross-validation experiment are shown in Figure 6.8. Although the scores are lower than those shown in the original 3-Player Soccer comparison in Chapter 6.1.5, the relative ordering of the algorithms remains the same. This result confirms that although Cascade-NEAT and RBF-NEAT generate solutions using local features, they still generalize well, and in particular better than

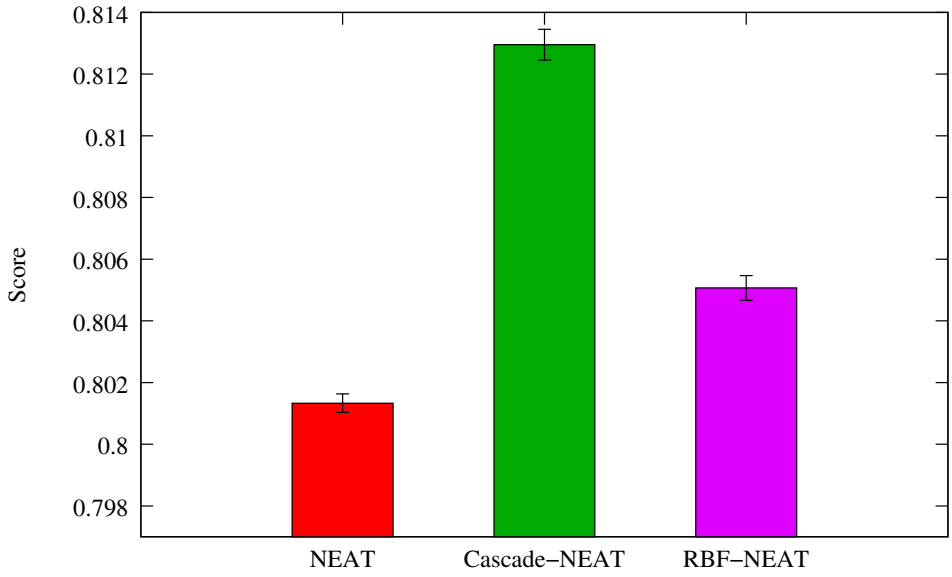


Figure 6.8: 10-fold cross-validation performance (mean and standard error) of the three learning algorithms on test data from the most difficult version of the 3-Player Soccer problem. Although scores are lower than the original 3-Player Soccer comparison, Cascade-NEAT and RBF-NEAT are still significantly better at generalizing than NEAT.

the standard NEAT algorithm.

6.3 Analysis of Performance

The results presented so far in this chapter show that Cascade-NEAT can significantly improve on NEAT's performance on a variety of fractured problems. RBF-NEAT also performs well, but it functions in a much more limited manner and seems to work well mainly when the dimensionality of the inputs is low. These results are encouraging, but there could be many explanations for this improvement in performance other than an affinity for creating local decision regions. In an effort to better understand the actual contribution of these algorithms, two analyses were performed: a maximizing variation study and an ablation study.

6.3.1 Maximizing variation

The first analysis was designed to evaluate how much variation the different learning algorithms can produce in an unrestricted setting. A problem (similar to that introduced in Chapter 4.3.1) was created where the only goal was to produce a "solution" that contained as much variation as possible.

Three different versions of this problem were created, each with a different number (one, two or three) of inputs. The input space in each case was uniformly divided into roughly 200 points. An evaluation consisted of evaluating a network on a each of these points and noting the output that was produced from the single output. The score for a network was the total variation of the discretized function that the network represented, calculated as described in Chapter 3.3.

The results from this experiment are shown in Figure 6.9. The amount of arbitrary variation that Cascade-NEAT is able to produce is significantly higher than that of the other algorithms. Interestingly enough, the amount of variation produced by RBF-NEAT is relatively large for a single input but falls off as the number

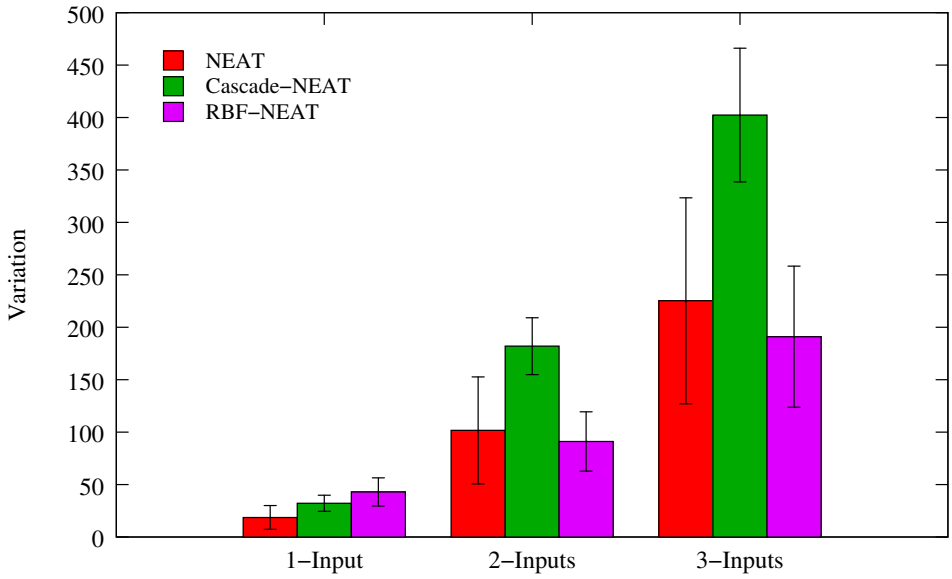


Figure 6.9: Performance of three learning algorithms on a problem where the goal is to produce a solution with as much variation as possible. Although RBF-NEAT performs well on the problem with the lowest dimensionality, Cascade-NEAT is able to produce significantly higher amounts of variation than the other algorithms on more complex problems.

of inputs increases, suggesting that RBF-NEAT is mainly effective at generating variation in low-dimensional settings.

These results provide additional evidence that constraining network construction is an effective method of increasing potential solution variation. Since the only metric that is measured in this experiment is the amount of variation that networks produce, it is clear that Cascade-NEAT is much more effective at producing networks with large amounts of variation.

6.3.2 Ablation study

In Chapter 4.3.2, an ablation study was performed on the best and worst networks produced by the NEAT algorithm on a fractured problem. Although it did not score well on average, occasionally NEAT produced networks capable of performing well

on the problem. Experiments showed that these superior networks were larger than the worst networks, and when ablated, they changed their output in a more local manner than did the worst networks. These results suggested that being able to create the necessary network structure to model local decision regions is critical in solving fractured problems, which led to the creation of the the RBF-NEAT and Cascade-NEAT algorithms.

Repeating this experiment with Cascade-NEAT is a natural way to test if the improved performance on the problems described above actually does stem from an ability to build larger networks with local decision regions. In this experiment, 25 champion networks from Cascade-NEAT and NEAT were systematically ablated 100 times. The resulting networks were measured for total change and locality of change compared to their parent networks, using the same process that is described above in Chapter 4.3.2. Figure 6.10 shows that ablated networks generated by Cascade-NEAT represent smaller and more local changes from their parents than do those produced by NEAT. These results suggest that the improved performance results from Cascade-NEAT's ability to build networks with local decision regions.

6.4 Empirical Analysis: Pole balancing

The experiments above suggest that biasing network construction towards structures that can perform local processing can improve performance on fractured problems significantly. But what about other types of problems? The benefits of biasing network construction that Cascade-NEAT uses could conceivably make it dominate the standard NEAT algorithm on all domains. The possibility that NEAT does not offer any advantages whatsoever over Cascade-NEAT, however unlikely, is worth examining.

One interesting problem on which NEAT has been thoroughly evaluated is double pole-balancing (Stanley, 2003). In this problem, the goal of the learning

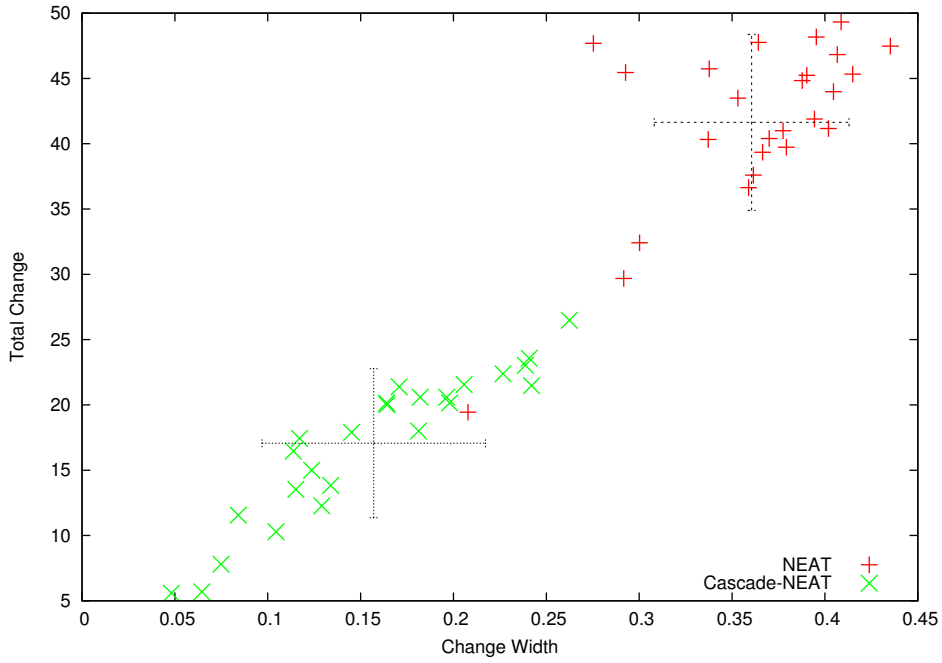


Figure 6.10: An ablation study on champion networks produced by NEAT and Cascade-NEAT. Networks found by Cascade-NEAT typically produce smaller and more local changes than do those produced by NEAT. This result lends credence to the hypothesis that Cascade-NEAT performs well on fractured problems because it has an affinity for creating local decision regions.

algorithm is to find a controller that can balance two poles of different lengths that are attached to a cart on a one-dimensional track. The controller receives input describing the position of the cart on the track and the angles of the two poles relative to the cart. In the Markov version of the problem, it also receives rates of change for these three variables; in the non-Markov version, it needs to estimate these rates by integrating information from previous states. The actions available to the controller provide an impulse to the cart that accelerates it in either direction on the track. Reward is proportional to the amount of time that the poles remain in the air, with the constraint that the cart must remain on a fixed section of the track.

The double pole-balancing problem is a classic reinforcement learning benchmark that has been used to gauge the performance of many learning algorithms. NEAT has performed well on this task in the past (Stanley and Miikkulainen, 2004a), which is not surprising because NEAT was built to solve this type of problem. Throughout its development, it was tested on pole-balancing, which no doubt impacted the various design decisions that were made as the algorithm was created (Stanley and Miikkulainen, 2002). As Figure 2.3 shows, it is possible to do well on double pole-balancing with a very small recurrent network. Since the NEAT algorithm starts with a population of minimal networks, it is well-prepared to solve problems that have small solutions.

All of these reasons suggest that it should be quite difficult to beat NEAT on the double pole-balancing problem. To evaluate this possibility, NEAT was compared to RBF-NEAT and Cascade-NEAT on both Markovian and non-Markovian implementations of the pole-balancing problem.

The results of the Markov comparison, shown in Figure 6.11, illustrate NEAT's prowess at performing well on this type of continuous control problem. Cascade-NEAT's ability to model local regions of the state space do not serve it well for

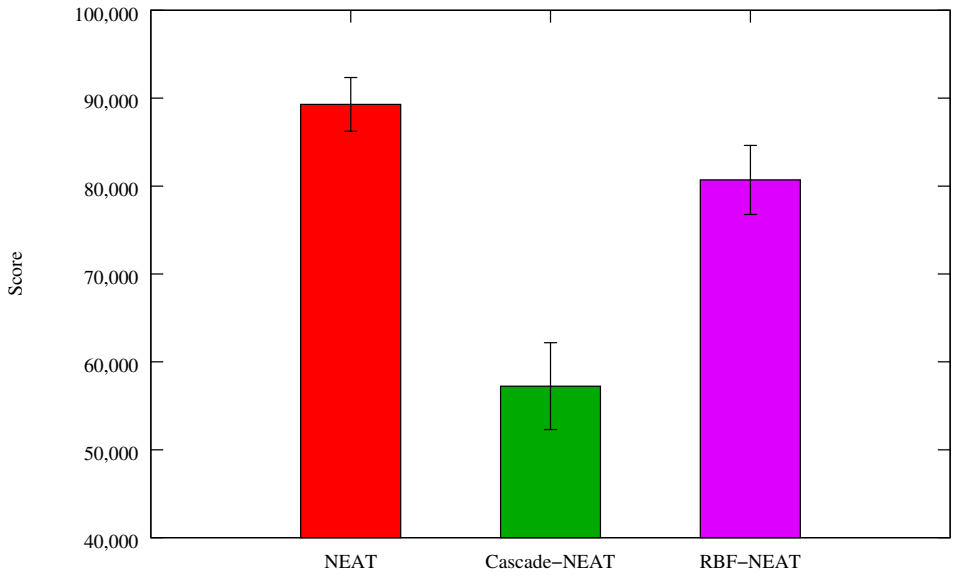


Figure 6.11: A comparison of NEAT, Cascade-NEAT, and RBF-NEAT on the Markovian double pole-balancing problem. Cascade-NEAT — which works well on fractured problems — performs poorly, while NEAT is able to consistently generate the best performance on this type of problem.

this problem. In fact, being forced to use the cascade architecture results in significantly lower performance. On the other hand, NEAT's affinity for generating small networks and making non-local changes to network output serves it well.

All of the analysis in this dissertation has focused on the definition of fracture proposed in Chapter 3, which is geared towards Markov problems. This restriction was only imposed to make analysis more manageable, however, and the results of this analysis – RBF-NEAT and Cascade-NEAT, which are general-purpose learning algorithms — might be applied more broadly. It is therefore interesting to examine how a non-Markov problem might highlight the differences between NEAT, Cascade-NEAT, and RBF-NEAT. The non-Markov double pole-balancing problem is quite similar to the pole-balancing problem described above, but the network receives no additional velocity inputs. This limitation makes the problem significantly more difficult, as determining the true state of the world involves integrating information

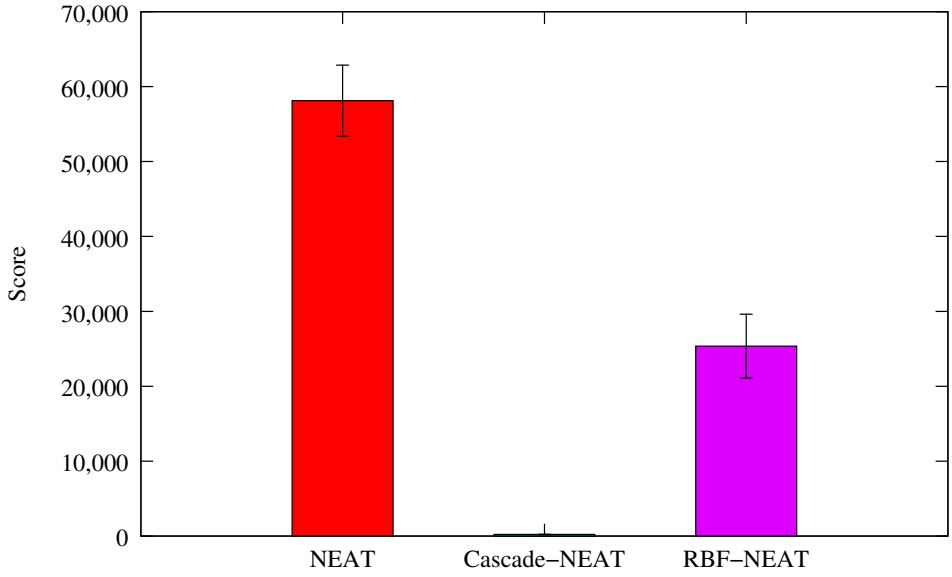


Figure 6.12: A comparison of NEAT, Cascade-NEAT, and RBF-NEAT on the non-Markovian double pole-balancing problem. Cascade-NEAT's inability to generate recurrent connectivity hurts it even more on this problem, whereas NEAT performs quite well.

over multiple timesteps. Figure 6.12 compares the performance of NEAT, Cascade-NEAT, and RBF-NEAT on this problem.

Not surprisingly, Cascade-NEAT and RBF-NEAT perform worse on this non-Markovian version of the problem compared to NEAT. Cascade-NEAT fails to find a single solution to the problem, whereas RBF-NEAT performs only half as well as NEAT — presumably due to the standard NEAT mutations that RBF-NEAT can employ. Cascade-NEAT's inability to produce recurrent connectivity patterns is almost certainly its downfall on this domain, where being able to track the state of the poles over multiple timesteps is the only obvious way to determine the direction in which they are moving.

6.5 Conclusion

This chapter provided an empirical evaluation of the hypothesis that it is beneficial to bias or constrain the search for network topology when learning in fractured problems. A variety of experiments showed that modified versions of NEAT were able to produce increased variation and higher scores on fractured problems. Analyses of these modified algorithms showed that they have an affinity for making local changes to candidate solutions.

However, while these modifications seem useful on fractured problems, the standard NEAT algorithm was shown to still offer the best performance on the standard benchmark domain of double pole-balancing. NEAT's incremental mutation operators, affinity for making non-local changes to network output, and ability to add recurrent connectivity allow it to explore the space of small networks quickly and efficiently. The next chapter explores the possibility of getting the best of both worlds by combining multiple approaches.

Chapter 7

An Integrated Algorithm: SNAP-NEAT

The experiments described so far showed that the standard NEAT algorithm has difficulty in solving fractured problems. By biasing and constraining the creation of network structure, algorithms like Cascade-NEAT can significantly improve performance on fractured problems. However, NEAT still yields the best performance on problems that can be solved with small, recurrent neural networks, like double pole balancing. Fortunately, because all of these methods share the same base design, it is possible to integrate them into a single algorithm that combines their strengths.

7.1 Combining Multiple Algorithms

The combined approach takes advantage of the fact that NEAT, RBF-NEAT, and Cascade-NEAT are almost completely identical except in their topological mutation strategy. The standard NEAT algorithm uses two topological mutation operators: add-link (between two unconnected nodes) and add-node (split a link into two links with a node between them). RBF-NEAT simply adds a third mutation operator,

add-RBF-node, which adds a special Gaussian basis-function node and connects it to inputs and outputs. In contrast, Cascade-NEAT uses only a single structural mutation operator, add-cascade-node, which adds a normal node that receives input from input and hidden nodes and which sends output to the output nodes. In addition, this operator freezes the previously-existing network structure to prevent the effective search space for connection weights from increasing too quickly.

One naive method of combining the strengths of these three algorithms is to simply make a new algorithm that chooses from all four structural mutations: add-link, add-node, add-RBF-node, and add-cascade-node. However, informal tests on various problems have found this approach to be lacking. Simply having access to a good mutation operator for a fractured problem (like add-cascade-node) does not mean that an algorithm will be able to effectively employ that operator. Ideally, the algorithm should be able to gauge the effectiveness of different operators and learn to select operators that are most suitable for the problem at hand.

7.1.1 Adaptive operator selection

The problem of choosing the correct mutation operators for a domain is known as adaptive operator selection (Goldberg, 1990, Barbosa and Sa, 2000, Julstrom, 1995, Thierens, 2005, DaCosta et al., 2008). The traditional and by far the simplest approach is to choose uniformly randomly between all operators. However, if certain operators are more useful than others, the selection of poor operators can limit learning performance. The goal of adaptive operator selection research is to make a more informed decision about which operators to choose.

Early research in adaptive operator selection collected statistics such as how frequently a chosen operator resulted in an improvement in score over its parents or resulted in a new best score for the entire population (Davis, 1989, Julstrom, 1995). In order to give credit to the operators that led to such individuals, the

estimated value of operators was propagated backwards from an individual to its parents. Some methods attempted to avoid the whole credit assignment problem by periodically re-calculating the value of all operators using only information from the current population (Tuson and Ross, 1998). After the value of the various operators was estimated using one of these methods, the probability of choosing an operator was calculated in a process known as Probability Matching (Goldberg, 1990, Barbosa and Sa, 2000). Such algorithms simply assign operator probabilities proportionally to expected value, while also enforcing certain minimum probabilities for each operator.

One of the more popular modern adaptive operator selection algorithms is Thieren's Adaptive Pursuit algorithm (Thierens, 2005). At every timestep, this algorithm attempts to identify the optimal probability for choosing operator o_i, with the goal of maximizing expected cumulative reward of the algorithm. It keeps estimates of the value Q_{o_i} for each operator, and then uses those estimates to weight the probability P_{o_i} of selecting each operator. Adaptive Pursuit is designed to respond quickly to changes in estimated operator value and emphasize selection of the highest-valued operator without completely ignoring other possible operators.

For example, given two operators o_1 and o_2, rewards $R_{o_1} = 10$ and $R_{o_2} = 9$, and a minimum probability $P_{min} = 0.1$, then Probability Matching will assign probabilities of $P_{o_1} = 0.52$ and $P_{o_2} = 0.48$ to the operators. It would be arguably preferable to have an algorithm that assigns probabilities of $P_{o_1} = 0.9$ and $P_{o_2} = 0.1$. Adaptive Pursuit achieves this goal by increasing the selection probability of the operator with the highest value, $o_* = argmax_i[Q_{o_i}]$:

$$P_{o_*}(t+1) = P_{o_*}(t) + \beta[P_{max} - P_{o_*}(t)] \tag{7.1}$$

while decreasing the probabilities of all other operators:

$$\forall o_i \neq o_* : P_{o_i}(t+1) = P_{o_i}(t) + \beta[P_{min} - P_{o_i}(t)] \tag{7.2}$$

In these equations, β is a free parameter that controls the rate at which these probabilities are updated. Another free parameter, α, serves a similar role in governing how fast the reward Q_{o_i} is updated. The value of P_{max} is constrained to be $1 - (K-1)P_{min}$, where K is the number of operators. These calculations effectively select the estimated optimal operator with probability P_{max}, while choosing uniformly among the other operators the rest of the time. This strategy allows Adaptive Pursuit to place much higher value on the single best operator than strategies like Probability Matching. Empirically, this decision has shown to improve performance, making Adaptive Pursuit an appealing choice for a NEAT-combination algorithm (Thierens, 2005, DaCosta et al., 2008). However, it is not necessarily straightforward to integrate Adaptive Pursuit with NEAT, as well be discussed next.

7.1.2 Continuous updates

Previous approaches to adaptive operator selection, including Adaptive Pursuit, estimate operator value immediately after application. For example, after an operator is chosen and used to create or update a member of the population, the resulting change in score is noted and applied to that operator. This approach, while certainly straightforward, is not necessarily appropriate for algorithms based on NEAT. One of the tenets of NEAT is that new structural mutations may require some time to be optimized before they become competitive with existing structures. The purpose of speciation in NEAT is to provide temporary shelter for new structures that arise in the population, giving them a fair chance to compete with structures that have had more time to be optimized. This concept of delayed evaluation has proven useful in NEAT (Stanley and Miikkulainen, 2004a), but conflicts slightly with the approach

taken by Adaptive Pursuit. Estimating the value of an operator immediately after application could result in an inaccurate estimate of the value of the operator.

An alternative method of estimating operator value is to keep track of which operator most recently affected each member of the population. As a given member of the population improves, the estimate of the value of the operator that most recently contributed to it will also be updated. In essence, every time an individual is updated (regardless of whether or not it was just modified by a structural mutation operator) an updated reward signal will be generated for the operator that most recently contributed to that individual. This process keeps operator values up-to-date with the current population, and also utilizes a much larger percentage of the information that the learning algorithm has available to it. Performing such continuous updates to operator values also fits nicely with the NEAT philosophy, where speciation is used to give new network topologies in the population a chance to compete.

7.1.3 Initial estimation

The standard Adaptive Pursuit algorithm uses a winner-take-all strategy to increase the likelihood of choosing the best operator at every timestep. This greedy approach is offset by a minimum probability P_{min} for each operator, which is designed to make it possible for the algorithm to change its operator selection strategy in the middle of learning. However, the winner-take-all strategy can still be sensitive to initial conditions.

If two operators have expected values that are close to each other, small differences in early evaluations can cause the Adaptive Pursuit algorithm to greedily choose the wrong operator. Such a mistake early in learning is not necessarily lethal — thanks to the minimum probabilities associated with each operator — but if the learning rate that governs how quickly probabilities can change is low, it can take a

while to recover from initial errors in probability estimation.

In order to better estimate the initial values P_{o_i} of all operators o_i, another modification to the Adaptive Pursuit algorithm was developed in this dissertation. The main idea is that the first N evaluations will serve as an evaluation period for all operators, wherein each operator will be evaluated an equal number of times. During this period, the probability P_{o_i} for each operator o_i will remain fixed and uniform. After the N evaluations have been completed, the information gained from those evaluations will be used to compute estimated values Q_{o_i} for each operator o_i. The algorithm then uses these initial value estimates to compute initial probability estimates P_{o_i} and resumes normal operation for the remaining evaluations.

Of course, since this initial evaluation period is used only to compute good estimates for operator values and does not attempt to take advantage of operators that appear to be performing well, such an approach could prove detrimental to learning. However, if it is important to start with good initial values for each operator, taking time for this initial evaluation could prove worthwhile. An empirical evaluation of how useful both continuous updates and an initial estimation period are is presented below.

7.2 SNAP-NEAT

SNAP-NEAT is a new version of the NEAT algorithm that uses Adaptive Pursuit to integrate the mutation operators from NEAT, Cascade-NEAT, and RBF-NEAT. The adaptive operator selection mechanisms from Adaptive Pursuit are used to choose between three different topological mutation operators: the NEAT operators, the RBF-NEAT operator, and the Cascade-NEAT operator. The two NEAT operators, add-node and add-link, are grouped together into a single operator for the purposes of estimating operator value and probability. When this operator is selected for actual use, a coin flip determines whether add-node or add-link is actually

run. This grouping forces the NEAT operators to change values in tandem.

SNAP-NEAT incorporates the two modifications discussed above, continuous updates and initial estimation. The period of initial estimation for most experiments is $N = 10000$, i.e. about one-fifth of the total learning time. During this initial estimation period, SNAP-NEAT cycles repeatedly between the three topological mutation types, noting the scores associated with each operator. When the initial estimation period ends, the value for each operator Q_{o_i} is initialized to one standard deviation above the mean of the values accumulated for o_i. In a manner similar to interval estimation, this method of initialization incorporates uncertainty about the true value for o_i (Kaelbling, 1993). For the remaining evaluations, a structural mutation operator o_i is selected according to its probability P_{o_i}, and both expected values Q_{o_i} and probabilities P_{o_i} are updated after each evaluation.

Thus, SNAP-NEAT uses a modified version of Adaptive Pursuit to make intelligent decisions about whether to favor NEAT, RBF-NEAT, or Cascade-NEAT for a given problem. The next section evaluates SNAP-NEAT on the problems described in Chapter 4 to determine its efficacy.

7.3 Empirical Evaluation

If SNAP-NEAT is successful, then it should be able to recognize which NEAT mutation strategy (NEAT, RBF-NEAT, or Cascade-NEAT) is required for a given problem. Selecting an appropriate strategy should improve SNAP-NEAT's performance relative to algorithms with a fixed strategy that is not suited to the given problem. In addition, examining how well the final probabilities for each operator match the best known algorithm can help determine how successful SNAP-NEAT was at selecting appropriate operators. This section revisits the problems presented in Chapter 4 to examine how well SNAP-NEAT can learn to use the correct operators.

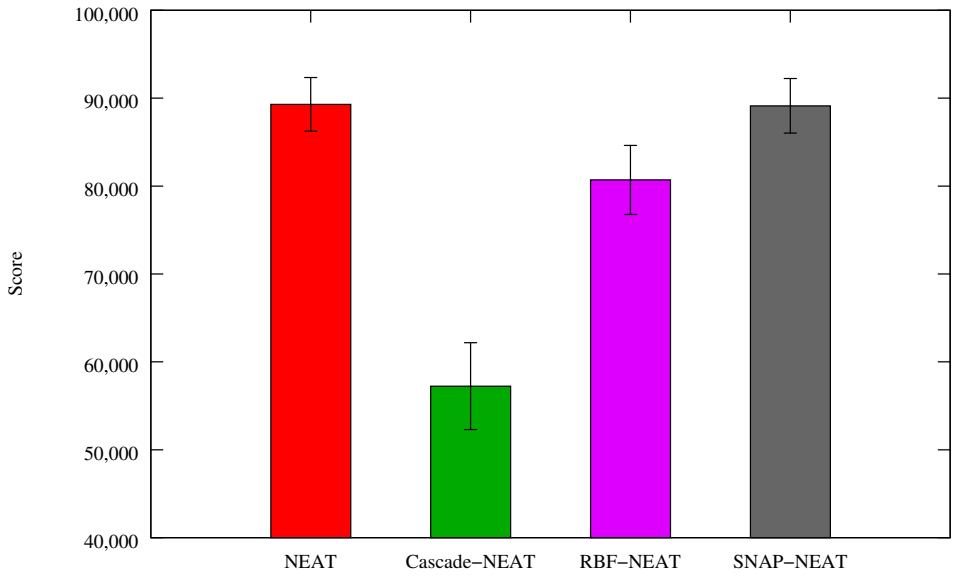


Figure 7.1: A comparison of NEAT, Cascade-NEAT, RBF-NEAT, and SNAP-NEAT on the Markov double pole-balancing problem. SNAP-NEAT learns that the standard NEAT mutation operators are most useful on this problem, giving it a performance comparable to that of NEAT.

7.3.1 Pole balancing

Figures 7.1 and 7.2 show results for the Markov and non-Markov double pole-balancing problems, comparing SNAP-NEAT to NEAT, Cascade-NEAT, and RBF-NEAT. On the Markov version of the problem, SNAP-NEAT's performance is indistinguishable from that of NEAT. On the non-Markov version of the problem, SNAP-NEAT takes a performance hit, apparently because it spends time considering the relatively useless add-cascade-node mutation. However, using the other operators allows it to perform quite well, achieving a level of performance near that of NEAT.

Examining the probabilities for each operator that are learned by SNAP-NEAT is another way to gauge the effectiveness of this approach. Since the standard NEAT algorithm generates the best performance on this problem, a successful

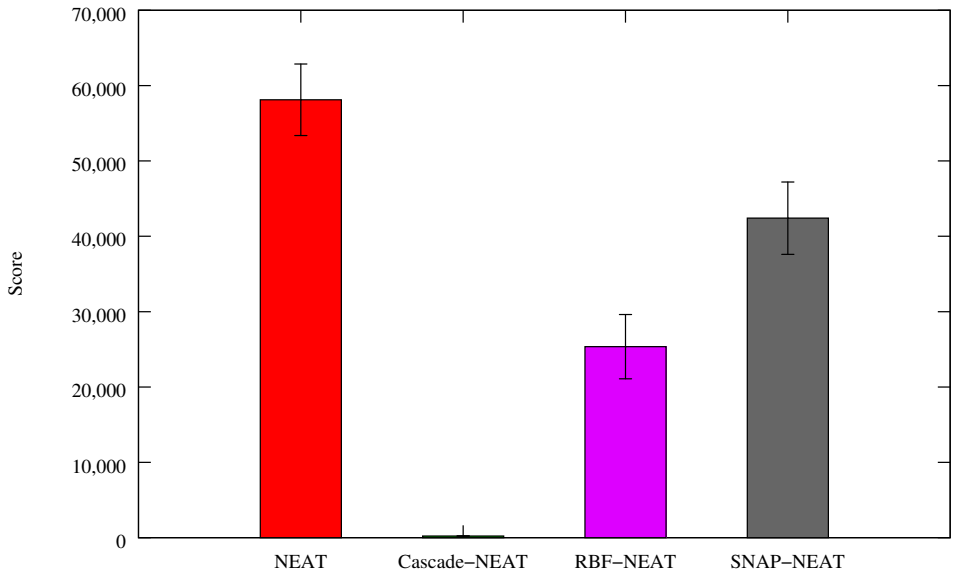


Figure 7.2: A comparison of several learning algorithms on the non-Markov double pole-balancing problem. SNAP-NEAT is able to achieve a performance level near that of NEAT, although its use of the add-RBF-node and add-cascade-node mutations limits its performance somewhat compared to NEAT.

operator selection algorithm should learn to favor the NEAT mutation operators. Figures 7.3 and 7.4 show the final average probabilities at the end of learning for successful runs of the SNAP-NEAT algorithm on the Markov and non-Markov versions of this problem.

In the Markov version of the problem, SNAP-NEAT learns to emphasize the NEAT mutations. This behavior is reasonable, given the high performance of the standard NEAT algorithm on this problem. Interestingly, in the non-Markov version of the problem, SNAP-NEAT does not learn to rely heavily on the NEAT mutations, instead striking a balance between NEAT and RBF-NEAT mutations. RBF-NEAT performs relatively well on this problem, so it is not surprising that SNAP-NEAT learns to use the add-RBF-node operator to some degree. The relatively low dimensionality of the input space makes any local processing more suitable for the add-RBF-node operator, which was shown on other problems to work best when

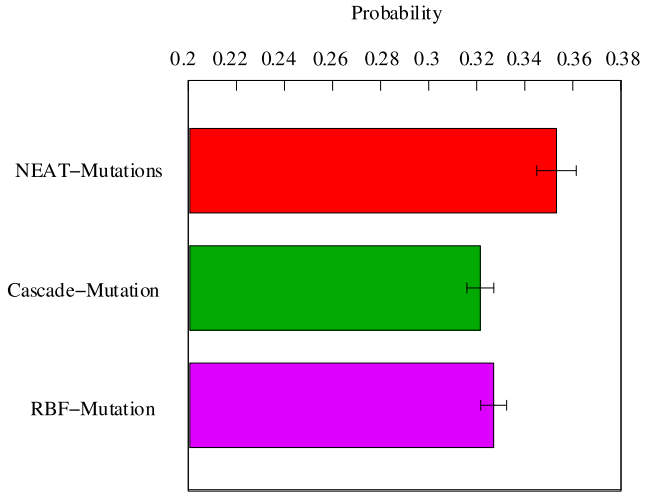


Figure 7.3: The learned operator probabilities for SNAP-NEAT in Markov double pole-balancing. SNAP-NEAT discovers that the NEAT mutations are the most useful for this problem, allowing it to perform at a level comparable to NEAT.

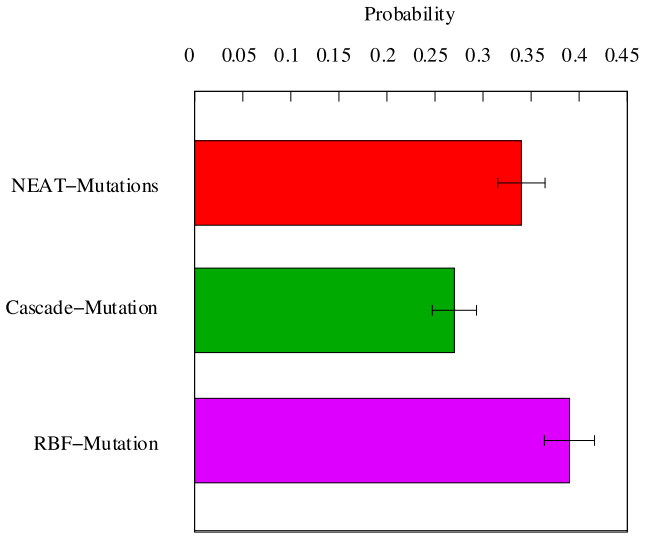


Figure 7.4: The learned operator probabilities for SNAP-NEAT in non-Markov double pole-balancing. SNAP-NEAT's ability to de-emphasize the add-cascade-node mutation allows it to find solutions almost as good as those found by NEAT. However, an over-reliance on the add-RBF-node operator, although not entirely unreasonable, results in lower performance than in the Markov version of the problem.

the number of inputs is low.

Perhaps the most important concept that it learned was to avoid using the add-cascade-node operator, which provides little utility on this problem (as witnessed by the low performance of Cascade-NEAT). The additional overhead of optimizing the many connections introduced by this operator outweigh the benefits of being able to isolate local regions of the input space. However, because SNAP-NEAT does not learn to de-emphasize the add-RBF-node operator to the same extent, its performance is somewhat limited compared to NEAT. Determining how to further improve the accuracy of SNAP-NEAT's operator evaluations is an interesting direction for future work that will be discussed further in Chapter 9.3.3.

7.3.2 N-Point classification

Perhaps because of its relatively low-dimensional input space, the 10-Point classification problem is one of the few where RBF-NEAT generated the best performance. Figure 7.5 compares the performance of SNAP-NEAT to the other algorithms on this problem, and Figure 7.6 shows the learned probabilities for SNAP-NEAT. SNAP-NEAT has learned to heavily favor the add-RBF-node mutation, confirming its ability to find the appropriate operator for this problem.

7.3.3 Function approximation

Figure 7.7 compares the performance of SNAP-NEAT to NEAT, Cascade-NEAT, and RBF-NEAT on the sine-wave function approximation problems introduced in Chapter 4.1.2. Unlike pole-balancing (at which NEAT excelled) or N-point classification (where RBF-NEAT did the best), these function approximation problems were best solved by Cascade-NEAT. It is therefore likely that if SNAP-NEAT is to work well on these problems, it will have to learn to utilize the add-cascade-node mutation.

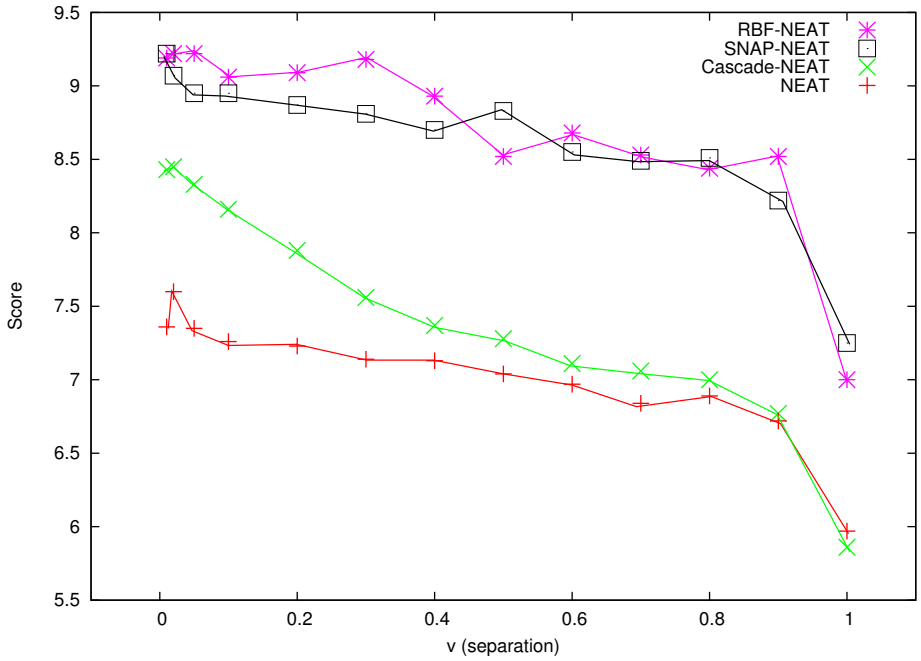


Figure 7.5: The performance of SNAP-NEAT on the 10-point classification problem. SNAP-NEAT is able to take advantage of the add-RBF-node mutation on this problem, giving it a score comparable to that of RBF-NEAT.

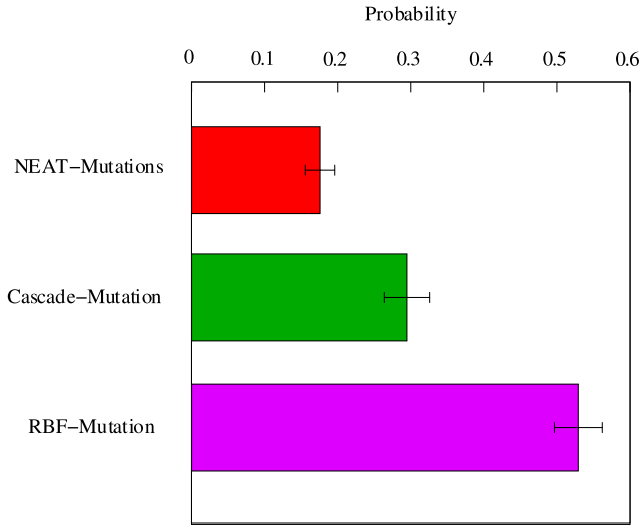


Figure 7.6: The probabilities learned by SNAP-NEAT for each operator for the 10-point classification problem when $\upsilon = 0.3$. RBF-NEAT offers the best performance for this problem, and SNAP-NEAT learns to significantly favor the add-RBF-node mutation.

Figure 7.7 shows that SNAP-NEAT consistently generates high performance on these function approximation problems. Interestingly enough, in several cases SNAP-NEAT actually outperforms Cascade-NEAT. This result shows the utility of an approach that combines multiple mutation operators rather than just selecting a single operator. Understanding and utilizing this possibility more systematically provides an interesting direction for future work.

The learned probabilities for several of the sine-wave function approximation problems are shown in Figure 7.8. Note the change in usage of the add-RBF-node and add-cascade-node operators as the level of fracture increases, showing SNAP-NEAT's ability to use different operators as the nature of the problem changes.

Figure 7.9 shows the performance of SNAP-NEAT on the second group of functions described in Chapter 4.1.2, while Figure 7.10 shows the learned probabilities for four of those problems. The results for these functions are similar to those of the sine-wave function approximation experiments. SNAP-NEAT consistently

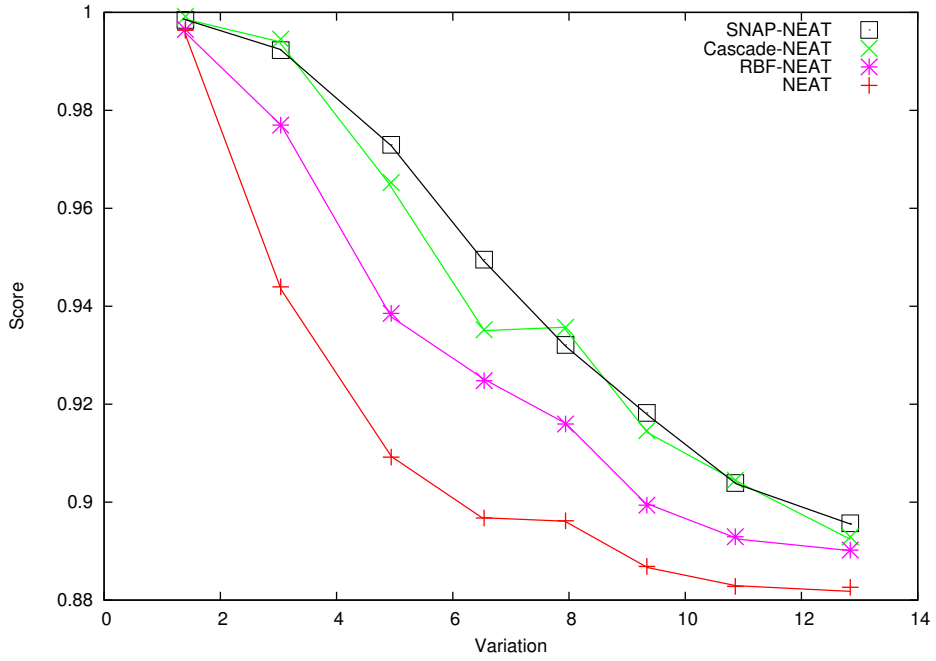


Figure 7.7: A comparison of several algorithms on the eight sine-wave function approximation problems. SNAP-NEAT performs surprisingly well, demonstrating its ability to use the add-cascade-node mutation when the need arises.

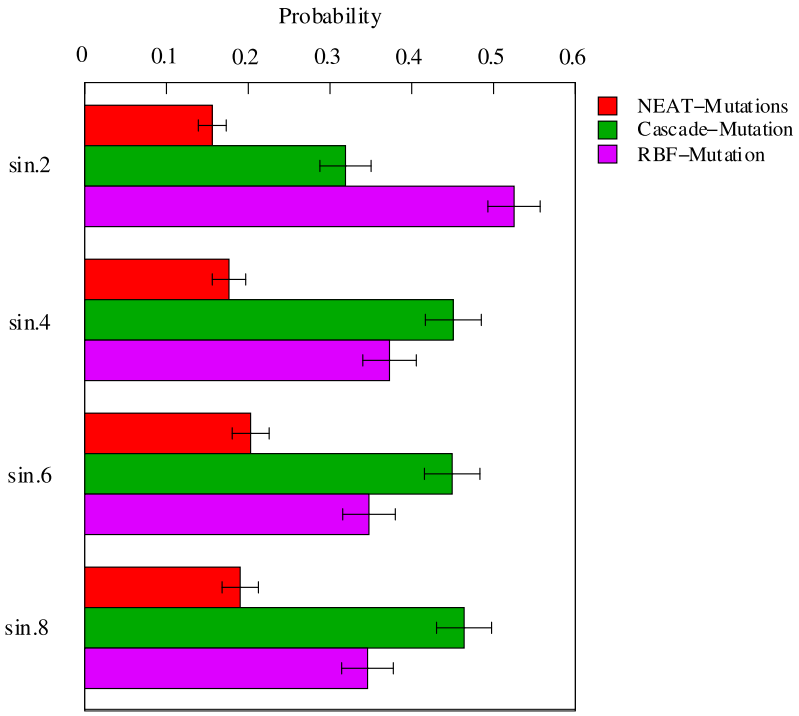


Figure 7.8: The learned operator distributions for SNAP-NEAT on several of the sine-wave function approximation problems. In general, as the level of fracture increases (from top to bottom), the utility of the add-cascade-node operator also increases.

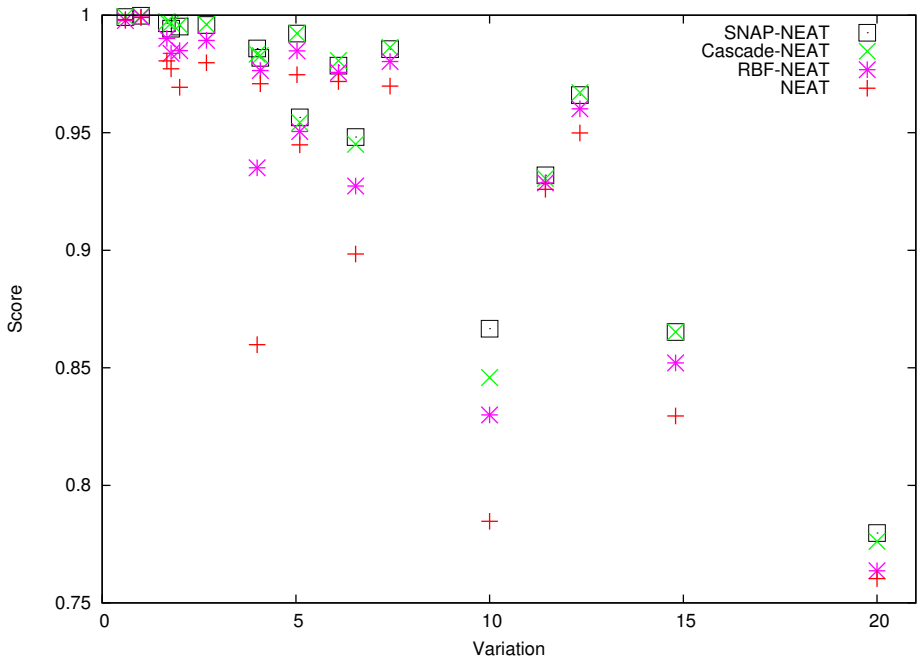


Figure 7.9: A comparison of several algorithms on the second group of function approximation problems. SNAP-NEAT consistently yields high performances, on par with those of Cascade-NEAT.

performs well, taking advantage of the add-cascade-node mutation with increasing frequency as the level of fracture increases.

7.3.4 Concentric spirals

Results for several versions of the concentric spirals problem are shown in Figure 7.11, followed by the learned probabilities in Figure 7.12. As with the function approximation experiments, these results confirm SNAP-NEAT's ability to perform well on fractured problems. As the level of fracture increases, SNAP-NEAT learns to become increasingly reliant on the add-cascade-node mutation.

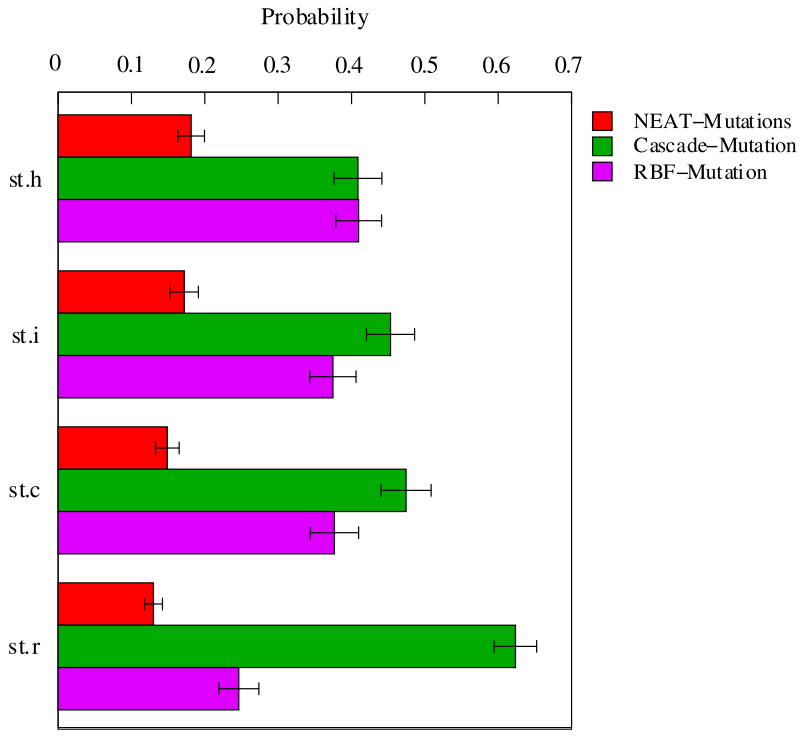


Figure 7.10: The operator probabilities learned by SNAP-NEAT for four of the function approximation problems, arranged from least fractured (top) to most fractured (bottom). SNAP-NEAT discovers different operator distributions depending on the level of fracture.

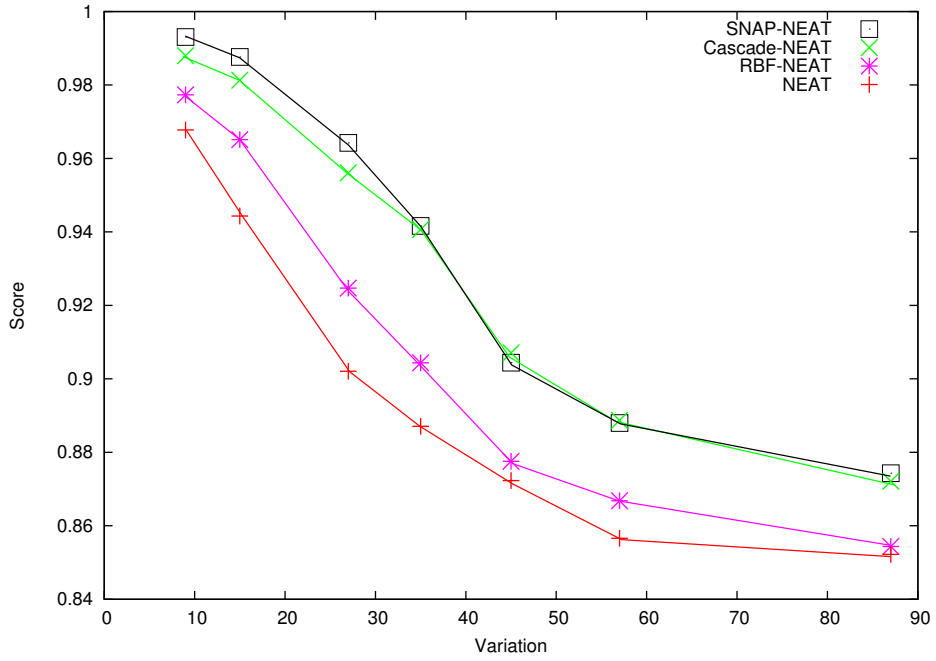


Figure 7.11: Performance of SNAP-NEAT on several versions of the concentric spirals problem. SNAP-NEAT is able to perform comparably to Cascade-NEAT.

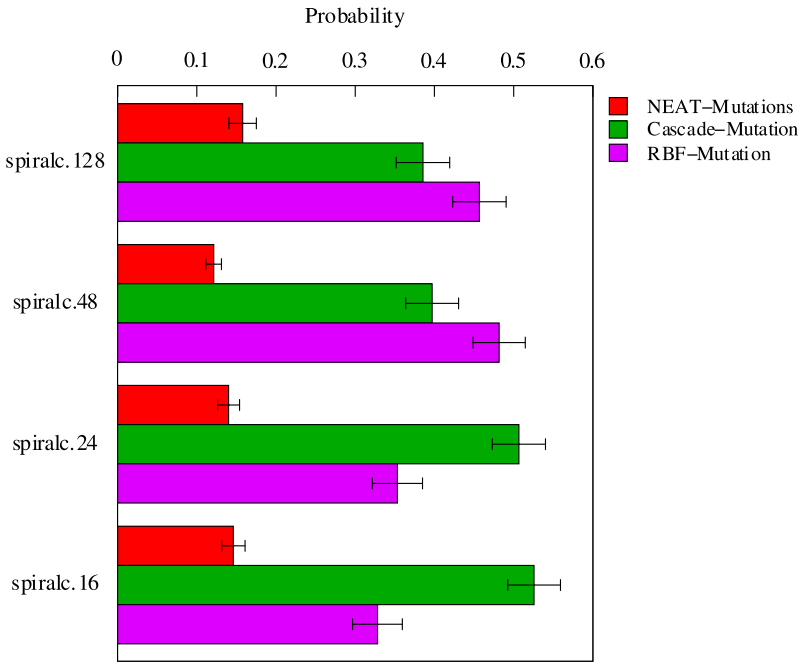


Figure 7.12: Operator probabilities learned by SNAP-NEAT for four versions of the concentric spirals problem, arranged from least fractured (top) to most fractured (bottom). As fracture increases, SNAP-NEAT increasingly favors the add-cascade-node operator.

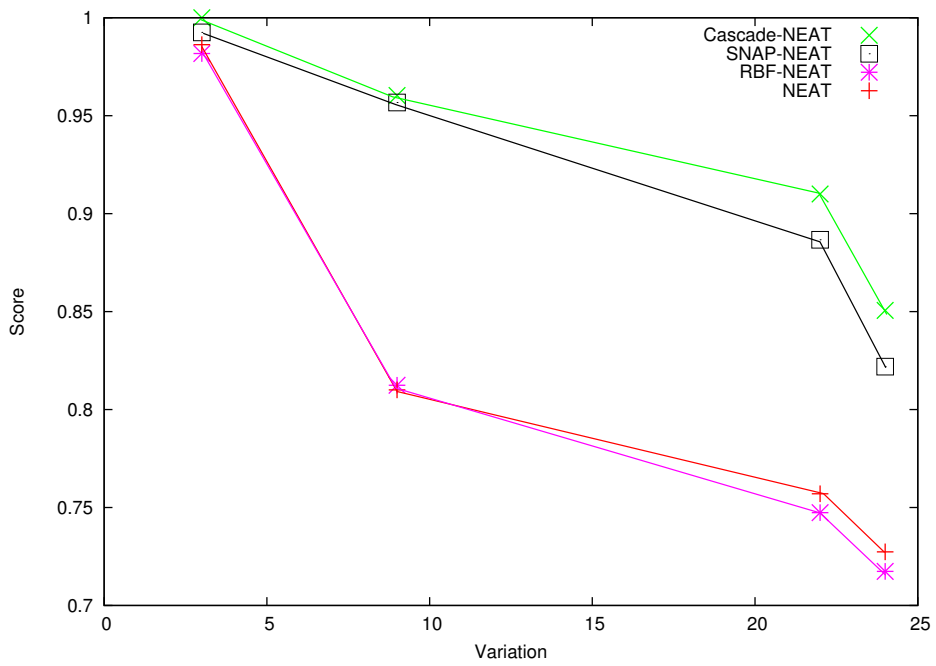


Figure 7.13: An evaluation of SNAP-NEAT on four versions of the multiplexer problem from Chapter 4.1.4. SNAP-NEAT's performance is near that of Cascade-NEAT.

7.3.5 Multiplexer

Results for four versions of the multiplexer problem (first introduced in Chapter 4.1.4) are shown in Figure 7.13, followed by learned probabilities in Figure 7.14. The utility of the Cascade-NEAT approach is exceedingly clear for this problem, and SNAP-NEAT correspondingly learns to heavily emphasize the add-cascade-node mutation on the more challenging versions of this problem. As before, this result demonstrates SNAP-NEAT's ability to favor one operator with near-exclusivity when the problem demands it.

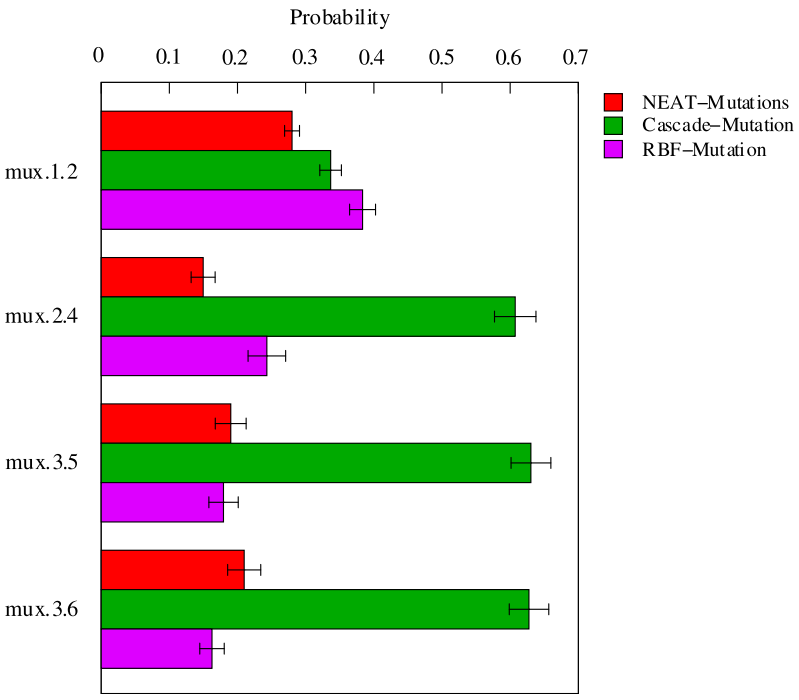


Figure 7.14: The probabilities learned by SNAP-NEAT for each operator for the four versions of the multiplexer problem. When the problem is relatively unfractured (the upper problems), the learned probabilities are similar. However, on the more difficult versions of this problem (near the bottom), SNAP-NEAT learns to rely heavily on the add-cascade-node mutation.

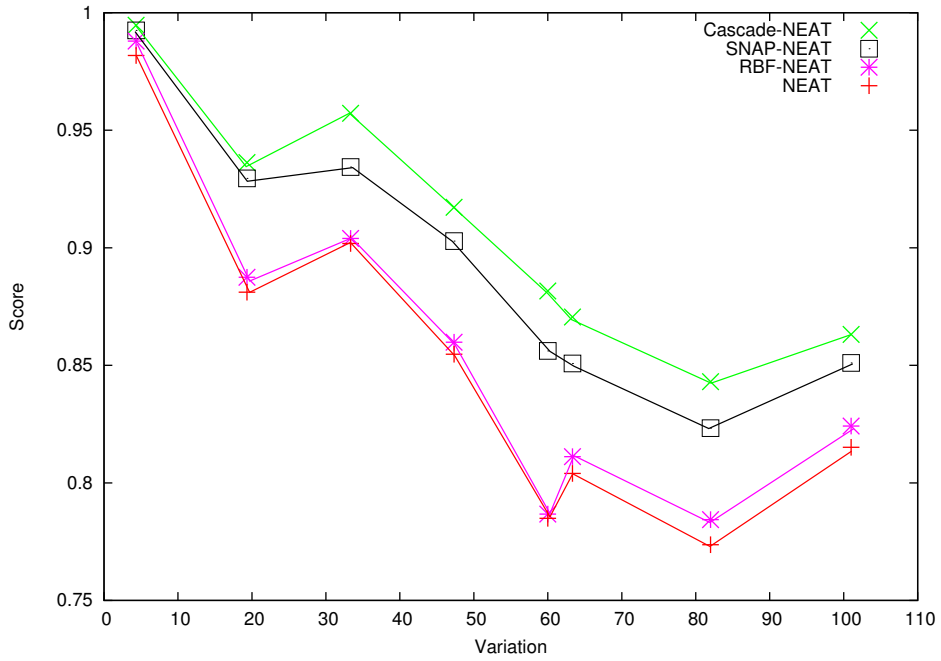


Figure 7.15: Performance of SNAP-NEAT on the 3-player soccer problems. The increased difficulty of these problems lowers SNAP-NEAT's performance, but it is still able to achieve results that are near those of Cascade-NEAT.

7.3.6 3-Player soccer

Results comparing SNAP-NEAT to the other algorithms on the 3-player soccer problem are shown in Figure 7.15. The increased difficulty of this problem lowers SNAP-NEAT's performance a bit, but it still offers levels of performance that are near that of Cascade-NEAT. The learned operator probabilities, shown in Figure 7.16, confirm that SNAP-NEAT learns to favor the add-cascade-node mutation for this problem.

7.3.7 Maximizing variation

When tasked with simply maximizing variation by any means possible, SNAP-NEAT performs quite well. Results comparing SNAP-NEAT to NEAT, RBF-NEAT, and

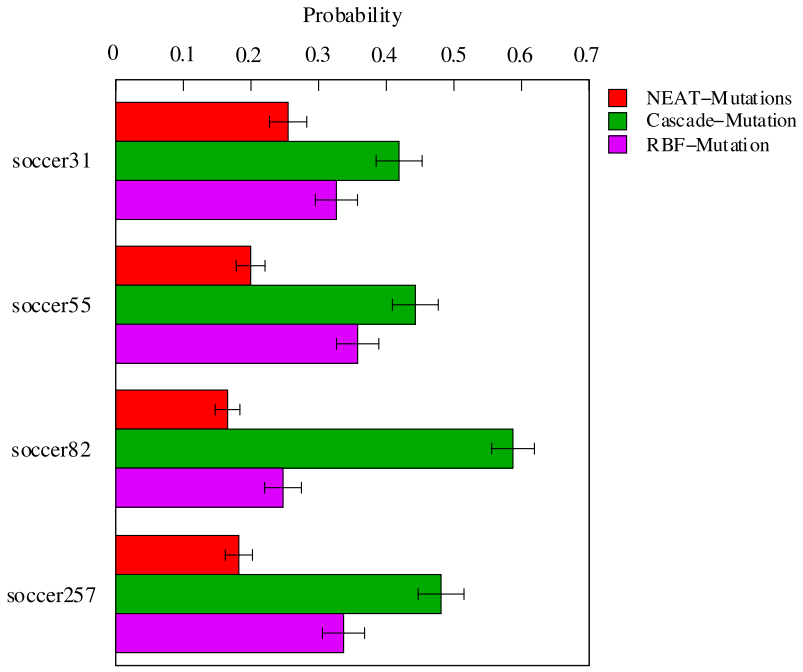


Figure 7.16: The operator probability distributions learned by SNAP-NEAT on four of the 3-player soccer problems. Despite the increased difficulty of the more fractured problems (near the bottom), SNAP-NEAT is still able to emphasize the appropriate operator.

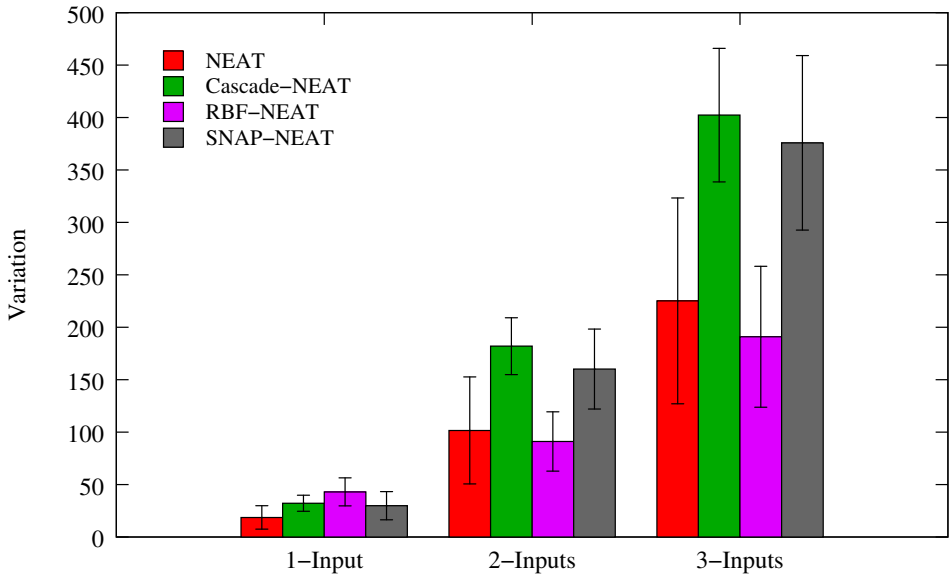


Figure 7.17: An evaluation of SNAP-NEAT on the maximizing-variation problem. SNAP-NEAT is able to generate high levels of variation — comparable to those of Cascade-NEAT — by learning which mutation operators to select.

Cascade-NEAT are shown in Figure 7.17. The operator probabilities learned by SNAP-NEAT are shown in Figure 7.18.

Interestingly enough, when SNAP-NEAT is faced with the 1-input version of this problem that RBF-NEAT seems to excel at, it learns to favor the add-RBF-node operator. When the input dimensionality increases, and Cascade-NEAT becomes the top performer, SNAP-NEAT learns to favor the add-cascade-node operator. These results offer more evidence of SNAP-NEAT's ability to correctly match operators to the problem at hand.

7.4 Comparison to Adaptive Pursuit

Recall that SNAP-NEAT augments the baseline Adaptive Pursuit algorithm with two modifications: continuous evaluations and a period of initial estimation. These changes make intuitive sense, and were designed to align Adaptive Pursuit with

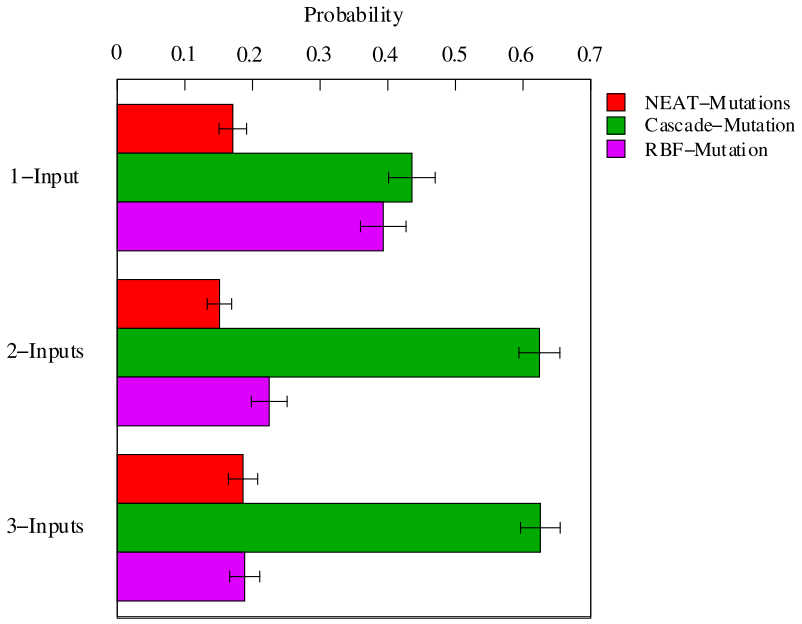


Figure 7.18: The operator probability distributions learned by SNAP-NEAT for the three variation maximization problems. SNAP-NEAT emphasizes the add-RBF-node mutation when the input dimensionality is small, but switches to the add-cascade-node mutation as dimensionality increases.

the core tenets of NEAT and make the integration of the two algorithms easier. However, it is worthwhile to examine how useful these modifications to the baseline Adaptive Pursuit algorithm actually are.

Figure 7.19 revisits the non-Markov version of the double pole-balancing problem described in Chapter 6.4. In addition to NEAT, Cascade-NEAT, and RBF-NEAT, SNAP-NEAT and three versions of the Adaptive Pursuit algorithm are also shown. The first version (labeled "AP") is the baseline Adaptive Pursuit algorithm with no modifications. The second two versions (labeled "AP+Init" and "AP+Cont") represent the baseline Adaptive Pursuit algorithm augmented with either the initialization period or continuous evaluation modification described above.

The results show that SNAP-NEAT is able to out-perform the standard Adaptive Pursuit algorithm, achieving a level of performance near that of NEAT. While modestly effective, the performance of Adaptive Pursuit can be significantly increased with the addition of continuous updates and an initialization period. Interestingly enough, either of these to modifications by themselves actually decreases performance, suggesting that a synergy exists between the two modifications.

Examining the probabilities for each operator that are learned by SNAP-NEAT is another way to gauge the effectiveness of this approach. Figure 7.20 compares these probabilities for successful runs of the SNAP-NEAT and baseline Adaptive Pursuit algorithms. Since the standard NEAT algorithm generates the best performance on this problem, a successful operator selection algorithm should learn to favor the NEAT mutation operator.

The differences in the learned operator probabilities are small, but significant. The Adaptive Pursuit algorithms do not learn to avoid the add-cascade-node mutation, despite the expectation that the NEAT mutations would be most useful for this problem. Since the Adaptive Pursuit algorithms perform poorly compared to SNAP-NEAT, it is reasonable to conclude that their distribution of operator

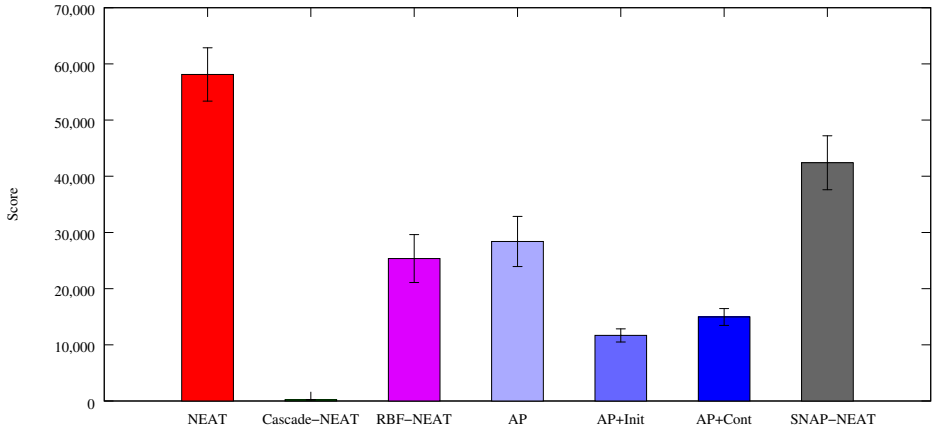


Figure 7.19: A comparison of several learning algorithms on the double pole-balancing problem. By using continuous evaluations and an initial estimation period, SNAP-NEAT is able to improve over the baseline Adaptive Pursuit algorithm and achieve a level of performance near that of NEAT.

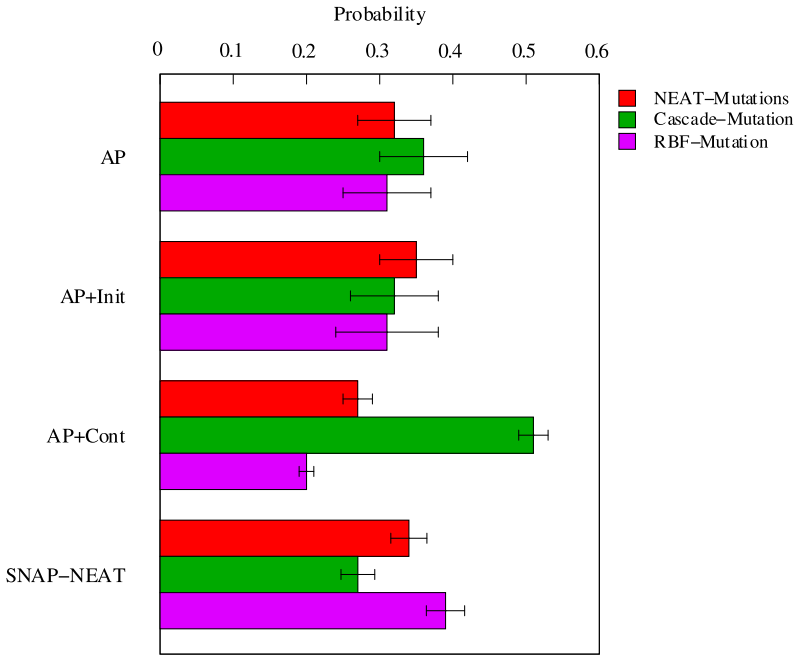


Figure 7.20: A comparison of the learned probabilities for three versions of the Adaptive Pursuit algorithm and SNAP-NEAT on the double pole-balancing problem. Although the differences are small, SNAP-NEAT's ability to de-emphasize the add-cascade-node mutation allows it to almost double the performance of Adaptive Pursuit.

probabilities is not appropriate for the pole-balancing problem. On the other hand, SNAP-NEAT learns to suppress the add-cascade-node mutation, instead favoring the NEAT and RBF-NEAT mutations. Using both continuous updates and a period of initial estimation allows SNAP-NEAT to discover this distribution of operator probabilities and outperform the baseline Adaptive Pursuit algorithm significantly.

In contrast to pole-balancing, where the standard NEAT algorithm works well, the multiplexer tasks described in Chapter 4.1.4 represent a spectrum of fractured problems more suitable for algorithms like Cascade-NEAT. Figure 7.21 compares the performance of SNAP-NEAT and Adaptive Pursuit variations to NEAT, RBF-NEAT, and Cascade-NEAT on the most fractured multiplexer problem. Figure 7.21 shows that SNAP-NEAT proves to be quite adept at the most difficult multiplexer problem. The baseline Adaptive Pursuit algorithm also does well — outperforming the standard NEAT algorithm — and the two individually modified Adaptive Pursuit algorithms offer a small increase in performance. However, when used together in the SNAP-NEAT algorithm, these modifications yield a significantly larger increase in performance.

The learned probabilities for this multiplexer problem are shown in Figure 7.22. The baseline Adaptive Pursuit algorithm has difficulty in favoring the add-cascade-node operator, which should be useful for this problem. When modified to include continuous evaluations, Adaptive Pursuit makes much better use of the cascade mutation. However, SNAP-NEAT favors the add-cascade-node operator even more heavily, and also learns to use the NEAT operators. This result demonstrates that SNAP-NEAT can learn to favor the appropriate operators for a given problem, whether it be fractured or unfractured.

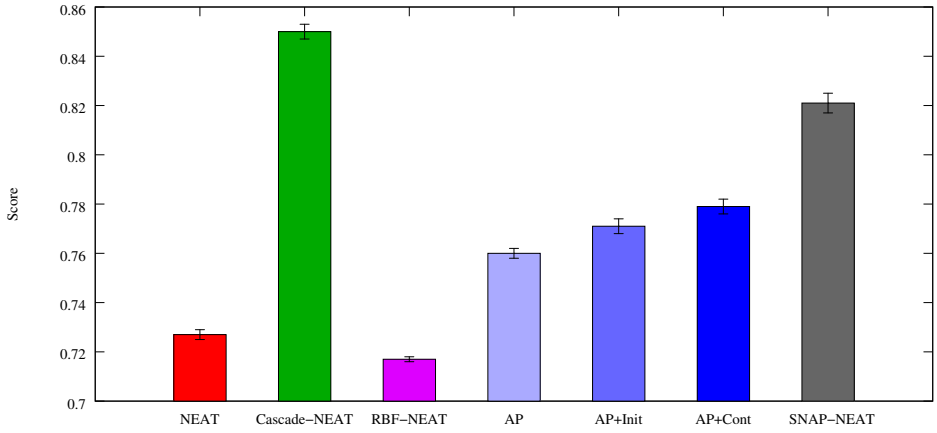


Figure 7.21: A comparison of Adaptive Pursuit and SNAP-NEAT on the most difficult version of the multiplexer problem from Chapter 4.1.4. SNAP-NEAT performs surprisingly well, demonstrating that it is important to accurately match correct operators with a given problem.

7.5 Conclusion

The results presented in this chapter show that it is possible to combine the benefits of NEAT, RBF-NEAT, and Cascade-NEAT into a single algorithm. Although there is some cost associated in exploring these different options, a combined algorithm like SNAP-NEAT yields good performance on a variety of problems regardless of the degree to which they are fractured. In addition, modifying the baseline Adaptive Pursuit algorithm to use continuous updates and an initial estimation period is shown to significantly improve performance.

However, the problems examined so far were chosen primarily because they are accessible and easy to analyze. In order to evaluate how well these modified neuroevolution algorithms perform on challenging and realistic high-level decision making problems, the next chapter presents an empirical comparison of these algorithms in the keepaway and half-field soccer domains.

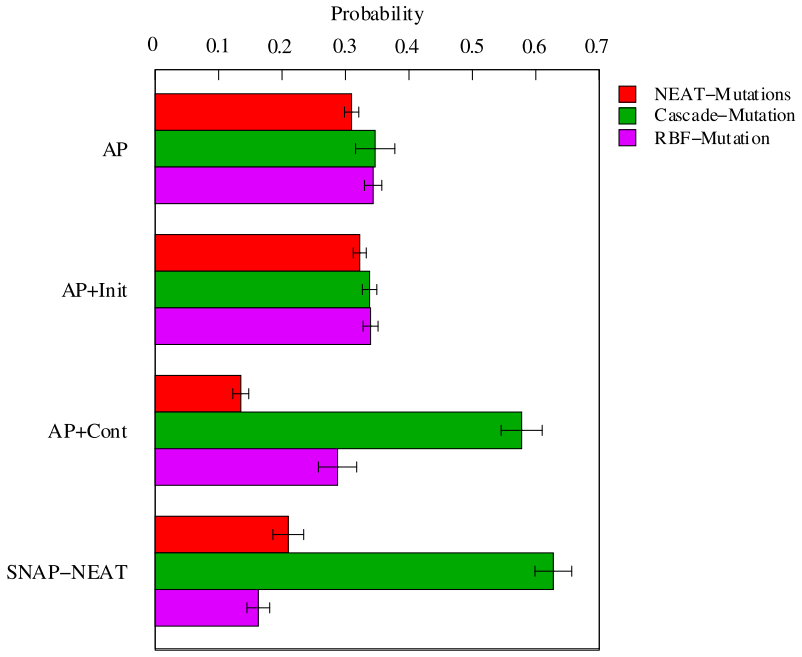


Figure 7.22: The learned operator distributions for the Adaptive Pursuit variants and SNAP-NEAT for the most challenging multiplexer problem. When modified to include continuous evaluations, Adaptive Pursuit learns to rely on the add-cascade-node operator. However, SNAP-NEAT also achieves this effect while maintaining a better mix of the other two operators, giving it a higher performance. This result shows that when used together, continuous evaluations and an initial evaluation period are important in estimating operator values accurately.

Chapter 8

Learning in High-level Decision Problems

The results presented in Chapters 4-7 provide evidence that "fractured" problems — problems where the correct actions vary frequently and abruptly between states — are difficult for constructive neural network algorithms like NEAT. While NEAT performs well on certain challenging problems like double pole-balancing, it lacks an ability to form local decision regions that are necessary to do well on fractured problems.

However, the problems tested in previous chapters are simple domains that were chosen primarily because they are easy to understand and analyze. An interesting question is to what extent do the same conclusions apply to complicated, modern reinforcement learning problems. Are "real" high-level decision problems fractured? Do the methods developed for dealing with fracture apply to them as well? This chapter addresses these questions by evaluating how well the algorithms developed in this dissertation perform and how much fracture there is in two challenging reinforcement learning problems: keepaway and half-field soccer.

8.1 Measuring Fracture in High-level Problems

Both keepaway and half-field soccer are challenging, continuous control problems that require the learner to develop a sophisticated high-level strategy. Intuitively, high-level problems like these could be considered more fractured than problems that require only low-level reasoning. High-level strategy frequently requires reasoning about a large area of state space, inside of which very disparate actions might be required. On the other hand, low-level behaviors operate on smaller sections of state space and require that actions are more continuous to avoid undesirable behaviors like thrashing. In half-field soccer, for example, tasks like dribbling the ball — which operates in a low-dimensional space and requires a fair amount of continuity to move smoothly and quickly to the ball — are pre-packaged and given to the player as atomic behaviors. Any local "smoothness" that might be required to execute a dribble is compressed into a discrete behavior that carries the agent into a new part of the state space. From the point of view of a high-level strategy, correct behaviors can change quickly and repeatedly because their effects are non-atomic.

Although it was possible to calculate fracture explicitly in the simple problems of the previous chapters, the complexity of keepaway and half-field soccer limits the depth of the analysis that can be performed. First, since optimal policies are not known for these problems, there is no way to measure the minimal variation required to solve the problems. Second, the large state spaces make it computationally intractable to sample a large number of points from better-performing policies. Third, the computational cost of evaluating policies makes it time consuming to perform multiple runs.

However, keepaway is less complex than half-field soccer and is therefore amenable to some analysis. By conducting multiple experiments that vary the number of states over which the learner must generalize, it becomes possible to increase the scope of the problem — which arguably should make it more fractured. The

decision space for both keepaway and half-field soccer can be visualized, illustrating how the correct action changes as the players move. These analyses are used to evaluate the hypothesis that one difficulty with high-level tasks is that they are fractured.

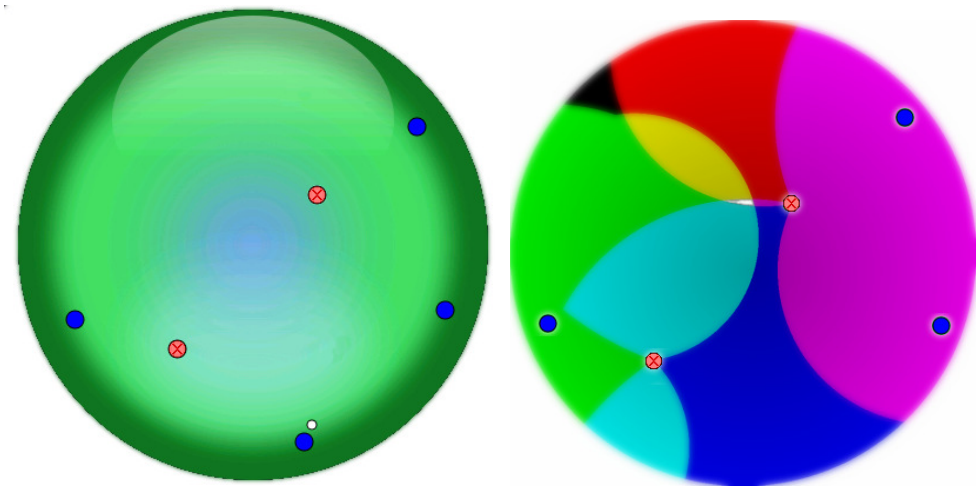
If decision spaces in high-level strategic tasks are indeed fractured, then algorithms like RBF-NEAT, Cascade-NEAT, and SNAP-NEAT should improve performance over the standard NEAT algorithm. Results similar to those obtained in previous experiments on fractured problems would offer credence to the hypothesis that high-level problems are fractured.

8.2 Keepaway

The first high-level task is the keepaway soccer problem (Stone et al., 2005, Whiteson et al., 2005, Stone et al., 2006). The particular version of keepaway soccer used in this dissertation includes four keepers that move at a limited speed competing against two faster takers in a circular field. The problem has ten continuous inputs and requires that the player choose between four high-level actions: pass to one of the other three keepers or hold the ball.

The goal is for the four keepers to prevent the two takers from gaining control of the ball in a bounded area. One feature that makes this particular version of keepaway difficult is that the takers can move five times faster than the keepers, which forces the keepers to develop a robust passing strategy instead of merely running with the ball. Figure 8.1 shows a typical state of a keepaway game and depicts the fractured decision regions of that state, similar to the illustration in Figure 1.2.

The takers behave according to a fixed, hand-coded algorithm that focuses on covering passing lanes and converging on the ball. The four keepers are controlled by a hierarchy of hand-coded and evolved behaviors (Figure 8.2). When a game



(a) (b)

Figure 8.1: (a) A starting configuration of players for the 4-versus-2 keepaway soccer problem. The four keepers (the darker players) attempt to keep the ball (shown in white) away from the two takers (the lighter players with crosses). (b) The color at each point p represents the set of available receivers to which a pass could be successfully sent if the player were at point p. Computing this set of possible receivers is difficult, especially if the player is on the left side of the field. In order to perform well in this task, the player must be able to model this fractured decision space.

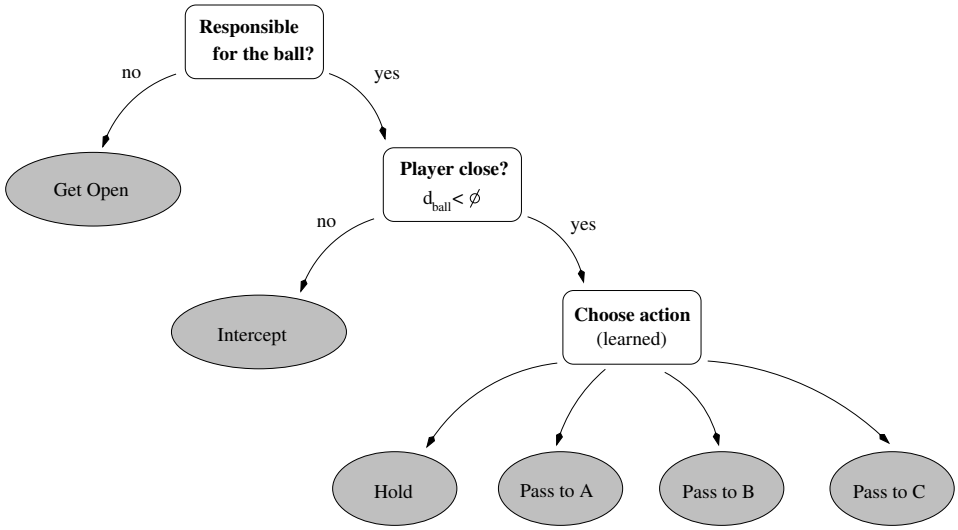


Figure 8.2: The decision tree used to control the keepers in the keepaway problem. Each player is controlled by a hierarchy of fixed and learned behaviors: The leaves of the tree are hand-coded low-level behaviors, whereas the action selection mechanism is learned by evolution.

starts, the keeper nearest the ball is made "responsible" for the ball. If this keeper is not close enough to the ball, it executes a pre-existing intercept behavior in an effort to get control of the ball. The keepers not responsible for the ball execute a pre-existing get-open behavior, designed to put the keepers in a good position to receive a pass.

When the responsible keeper has control of the ball (defined by being within ϕ meters of it) it must choose between executing a pre-existing hold behavior or attempting a pass to one of its three teammates. The goal of learning is to make the appropriate decision given the state of the game at this point. Note that making the correct decision at this point is quite difficult; while the decisions for other parts of the behavior are intuitive and relatively easy to encode with fixed rules (e.g. if no players are close to the ball, the closest should intercept it) this decision has no obvious correct answer.

To make this decision, the network controlling the responsible keeper receives

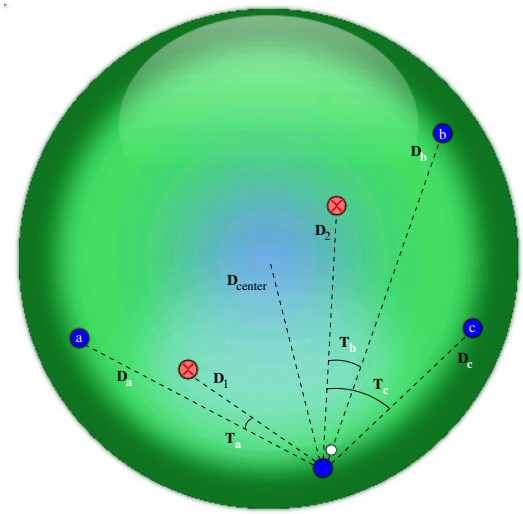


Figure 8.3: A graphical depiction of the state that a keeper observes when making decisions. The inputs represent distances and angles between teammates and opponents, normalized into the range $[0, 1]$.

ten continuous inputs (Figure 8.3). The first input describes the keeper's distance from the center of the field. The network also receives three inputs for each of the three teammates: the distance to that teammate, the angle between that teammate and the nearest taker, and the distance to the nearest taker. All angles and distances are normalized to the range $[0, 1]$. The network has one output for each of the four possible actions (hold or pass to one of the three teammates). The output with the highest activation is interpreted as the keeper's action.

If the responsible keeper chooses to pass, the keeper receiving the pass is designated the responsible keeper. After initiating the pass, the original keeper begins executing the get-open behavior.

Each network was evaluated from $\tau = 30$ different randomly chosen initial configurations of takers and keepers. In each configuration, both takers and keepers are assigned random locations inside the field, and the ball is initially placed near one of the keepers. Each of the players executes the appropriate hand-coded behavior,

and the current network being evaluated is used to select an action whenever the keeper responsible for the ball needs to choose between holding and passing. The game is allowed to proceed until a timeout is reached, the ball goes out of bounds, or a taker achieves control of the ball (by getting within ϕ meters of it). The score for a single game is the number of timesteps that the game takes. The overall score for the network is the sum of the scores for all τ games.

An empirical comparison was performed between five learning algorithms (SNAP-NEAT, NEAT, Cascade-NEAT, RBF-NEAT, and the linear baseline algorithm) as well as a hand-coded solution for the keepaway soccer problem. The hand-coded solution attempted to reconstruct the complete state of the game using the current state input, and then ran a short simulation forward in time to calculate the odds of successfully passing to each teammate. Because the state information is limited, however, reconstructing the true state of play at any given time is difficult and prone to error. Furthermore, best-guess heuristics must be employed to predict what actions the opponents will execute, which introduces more possibility for error.

Figure 8.4 compares these six approaches on the keepaway problem. NEAT was able to offer moderate improvement over the hand-coded policy, but Cascade-NEAT offers the highest performance by a wide margin. SNAP-NEAT captures some of the performance of Cascade-NEAT as well, performing significantly better than the other algorithms in this comparison. Animations of the best learned and hand-coded policies can be seen at `http://nn.cs.utexas.edu/?fracture`.

One method of varying the amount of fracture in the keepaway domain is to change τ, the number of initial states on which each network is evaluated. Reducing the number of starting states should reduce variation, making the problem easier to solve. Intuitively, this modification has the effect of reducing the area over which the network must generalize, resulting in a simpler function that the network can approximate more easily. In other words, as the number of required states decreases,

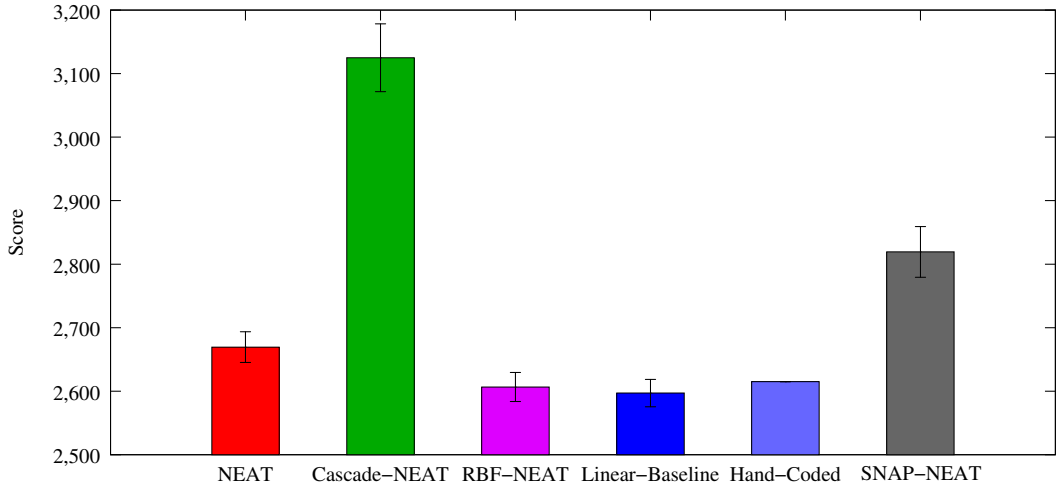


Figure 8.4: Comparison of several learning algorithms and a hand-coded solution in the keepaway soccer problem with 30 starting states. While NEAT is able to slightly improve on the hand-coded behavior, Cascade-NEAT offers the best performance by a wide margin. SNAP-NEAT provides some of the performance of Cascade-NEAT, performing significantly better than the other algorithms. Animations of the best learned policies can be seen at `http://nn.cs.utexas.edu/?fracture`.

it should become easier to solve the problem with a relatively simple mapping from states to actions.

In general, versions of the keepaway problem with fewer starting states are indeed easier to solve (Figure 8.5). As the number of starting states increases, the superior performance of Cascade-NEAT becomes more pronounced. These results support the hypothesis that problems become increasingly fractured as the scope of learning increases, and that Cascade-NEAT is much more adept at solving these fractured problems. Although its performance is not as high as that of Cascade-NEAT, SNAP-NEAT offers a compromise between the performance of NEAT and Cascade-NEAT.

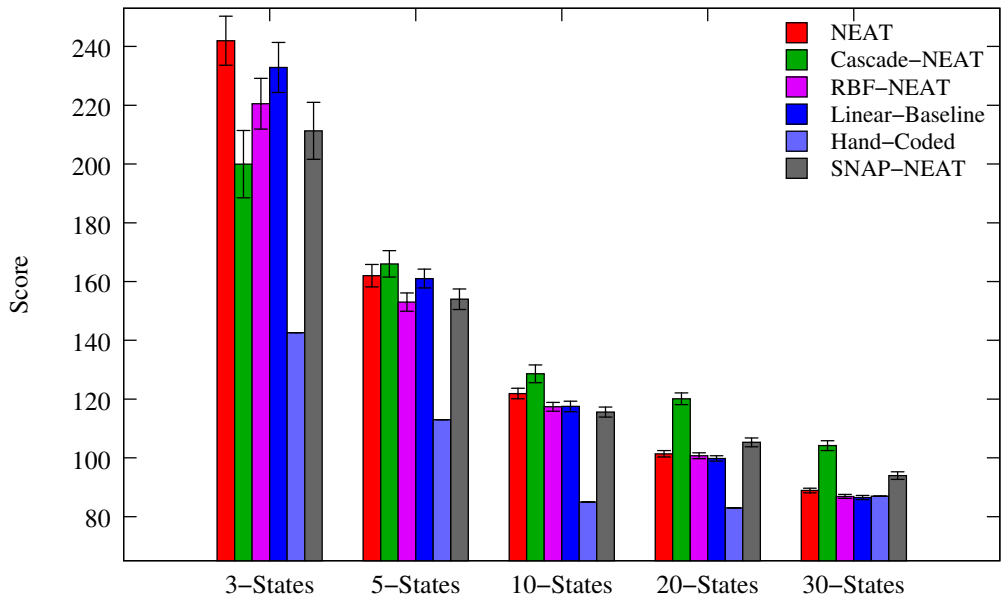


Figure 8.5: Measuring the effect of the number of starting states on learning performance. As the number of starting states increases, the relative performance gain provided by Cascade-NEAT increases. This result suggests that the utility of Cascade-NEAT increases with problem fracture, making it a good candidate for learning high-level decision-making tasks.

8.3 Half-field Soccer

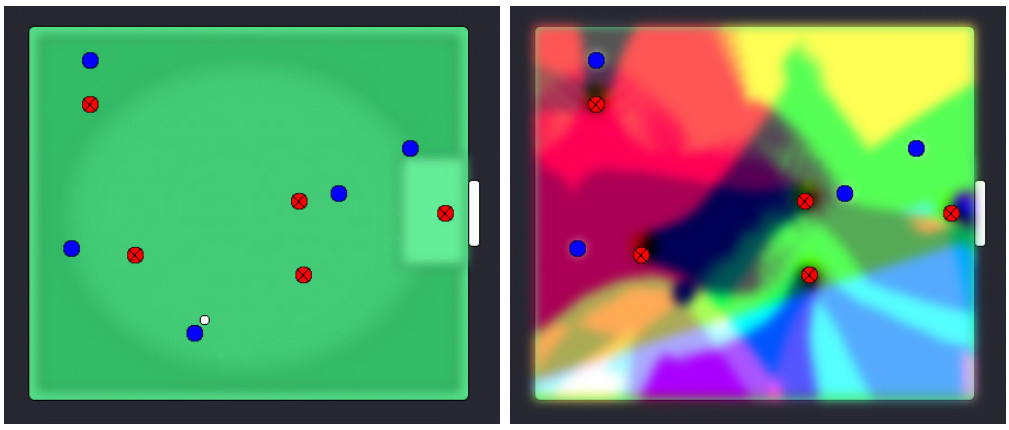
Half-field soccer is a more complex benchmark problem that introduces more players and an additional objective of scoring goals (Kalyanakrishnan et al., 2007). Because of this increased complexity, it is difficult to analyze half-field soccer computationally in even the limited manner that was possible for keepaway. However, the increased complexity also makes half-field soccer an interesting and challenging domain to evaluate the efficacy of RBF-NEAT, Cascade-NEAT, and SNAP-NEAT.

The version of half-field soccer used in this dissertation features five offenders, five defenders, and a ball. A game starts with a random configuration of players on a rectangular field, as shown in Figure 8.6. One of the defenders is designated as the "goalie", and is tasked with defending a goal on the right side of the field. The other defenders follow a hand-coded behavior designed to cover the field, prevent goals, and intercept the ball from the offending team.

In a manner similar to keepaway, the offensive team attempts to keep the ball away from a set of defenders. However, this objective is now secondary; the main source of reward for the offense is to score goals on the defending team.

As in keepaway, the offenders are controlled by a hierarchy of hand-coded and learned behaviors (Figure 8.7). When a game starts, the offensive player nearest to the ball is designated as responsible for the ball. If this player is not close enough to the ball, it executes a pre-existing intercept behavior in an effort to get control of the ball. The other offensive players not responsible for the ball execute a pre-existing get-open behavior, designed to put them in good positions to both receive passes and to score goals.

However, when the responsible offender has control of the ball (defined by being within ϕ meters of the ball) it must choose between pre-existing behaviors of holding the ball, kicking the ball at the goal, or attempting a pass to one of its four teammates. The goal of learning is to make the appropriate decision given the state



(a) (b)

Figure 8.6: (a) An example configuration of the half-field soccer domain, where five offenders (darker players) attempt to score goals on five defenders (lighter players with crosses). (b) An illustration of which actions would be successful for an offender with the ball at various points on the field. Each color represents a different set of actions (chosen from holding, shooting on goal, or passing to one of four teammates) that, if executed, would not immediately result in the end of an episode. The additional players and the objective of scoring goals makes half-field soccer even more challenging than the keepaway problem.

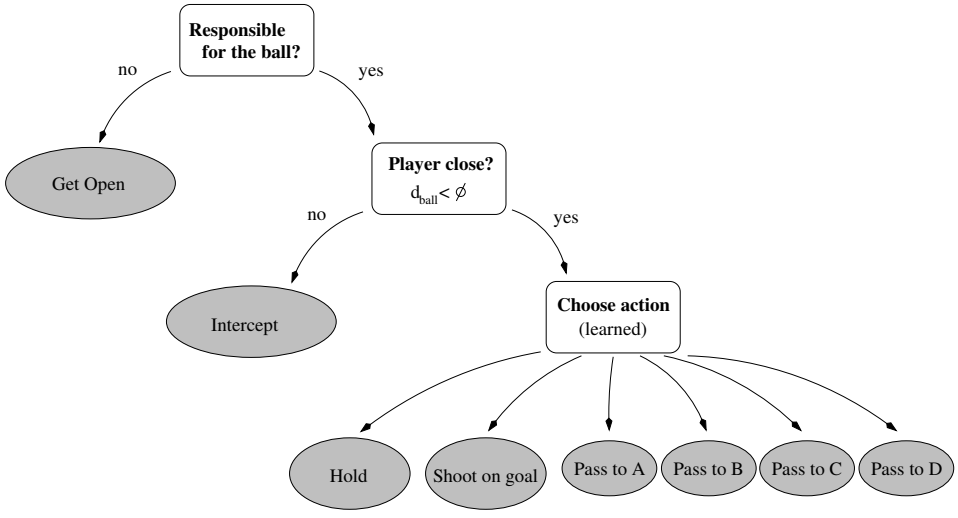


Figure 8.7: The decision tree used to control the offensive players in the half-field soccer problem. Each player is controlled by a hierarchy of hand-coded and learned behaviors.

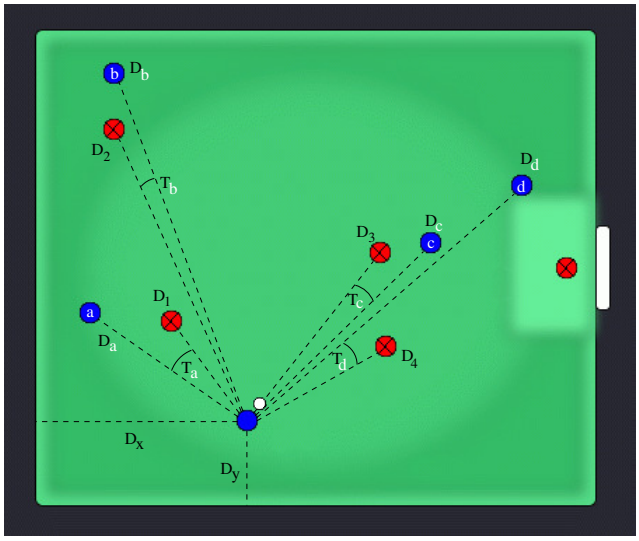


Figure 8.8: A graphical depiction of the 14 state variables that an offensive player observes when making decisions in the half-field soccer problem. The inputs represent position on the field as well as distances and angles between teammates and opponents, normalized into the range $[0, 1]$.

of the game at this point. As in keepaway, this decision is difficult and is therefore a good test for learning algorithms.

To make this decision, the network controlling the responsible offender receives 14 continuous inputs (Figure 8.8). The first two inputs describe the player's position on the field. The network also receives three inputs for each of its four teammates: the distance to that teammate, the angle between that teammate and the nearest taker, and the distance to that nearest taker. All angles and distances are normalized to the range $[0, 1]$. The network has one output for each possible action (hold, shoot, or pass to one of the four teammates). The output with the highest activation is interpreted as the offender's action.

If the offender chooses to pass, the teammate receiving the pass is designated the new responsible offender. After initiating the pass, the original offender begins executing the get-open behavior.

Each network was evaluated in $\tau = 50$ different randomly chosen initial configurations of takers and keepers. In each configuration, the ball is initially placed near one of the offensive players. Each of the players executes the appropriate hand-coded behavior. When the player responsible for the ball needs to choose between holding, shooting, and passing, the current network is used to select an action. The game is allowed to proceed until a timeout is reached (after 1000 timesteps), the ball goes out of bounds, a goal is scored, or a taker achieves control of the ball (by getting within ϕ meters of it). The score for a single game is the number of timesteps that the game takes, or, if a goal is scored, a fixed reward of 10,000. The overall score for the network is the sum of the scores for all τ games.

In order to evaluate the performance of the NEAT-related algorithms, several additional learning algorithms that have shown promise on domains like half-field soccer were also evaluated. The first algorithm is the standard reinforcement learning approach known as SARSA, which was highlighted as the best learning approach in the original half-field soccer paper (Kalyanakrishnan et al., 2007). This type of classic reinforcement learning algorithm has been shown to work well on challenging problems like keepaway and half-field soccer (Stone et al., 2005). The version used for this comparison employs the same system of shared updates described in Kalyanakrishnan's work, which was found to offer better performance than the baseline SARSA approach used for keepaway. Similarly, a CMAC function approximator was used to model the value function during learning, because it also was shown to generate the best results in previous work.

In addition to the NEAT variants, the ESP neuroevolution algorithm was evaluated on the half-field soccer problem. ESP has been shown to be effective in the past at generating controls for non-linear control tasks such as rocket stabilization (Gomez et al., 2006), often outperforming other reinforcement learning approaches. Since ESP relies on a fixed network topology to be chosen a priori by

the experimenter, several different recurrent and non-recurrent network topologies were examined. The best approach ended up using a network with five hidden nodes with fully recurrent connections.

The final algorithm that was compared with NEAT was a hand-coded policy optimized by a vanilla genetic algorithm. This policy is based on linear combinations of evolved parameters to make decisions about the chances of success for shooting, holding, and passing. The genetic algorithm used to evolve these parameters was the same algorithm that was used to optimize the weights of networks in the NEAT variants.

The high computational cost of running simulations of half-field soccer made it prohibitive to perform experiments with different numbers of starting states, like those presented above for the keepaway problem. However, it was possible to gather enough data for the 50-start state case to see significant patterns emerge. Figure 8.9 shows the scores for NEAT, Cascade-NEAT, the linear baseline algorithm, RBF-NEAT, SARSA+CMAC, ESP, and the hand-coded/GA approaches, averaged over 100 runs.

NEAT is able to do reasonably well on this problem, outperforming ESP and the hand-coded/GA approaches. SNAP-NEAT performs statistically as well as the SARSA+CMAC approach. However, Cascade-NEAT generates the highest level of performance by a clear margin. This result suggests that combining NEAT with an ability to model local decision regions is a promising approach for learning high-level strategies.

8.4 Conclusion

This chapter evaluated the NEAT variants on two high-level reinforcement learning domains, keepaway and half-field soccer. Although the detailed analysis performed on simpler problems is not feasible for these more complicated domains, the similar

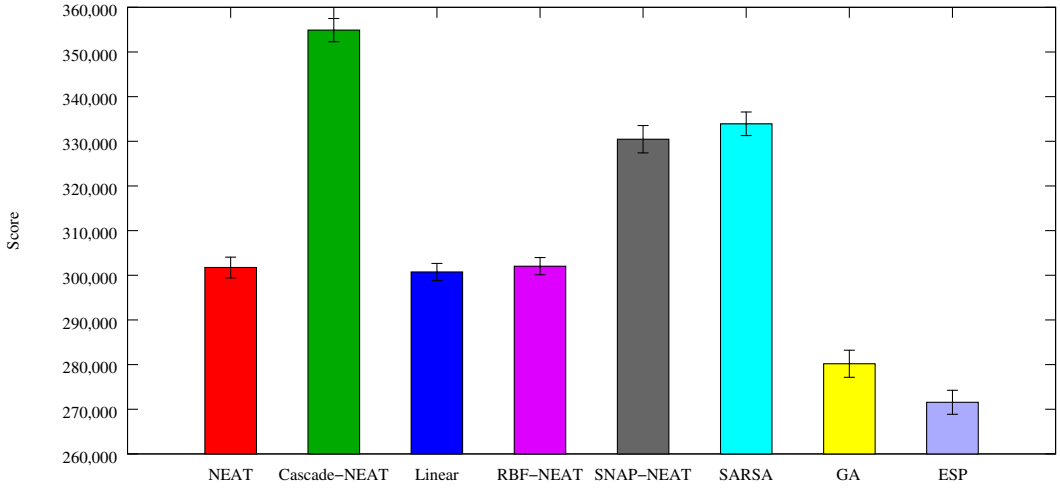


Figure 8.9: A comparison of several learning algorithms on the half-field soccer problem. Cascade-NEAT and SNAP-NEAT are among the best-performing approaches, providing evidence that combining NEAT with the ability to model local decision regions is a powerful approach for learning high-level strategy.

pattern of results suggests that learning performance can be improved when specific attention is focused on modeling local decision regions. Cascade-NEAT — which is heavily biased towards solving fractured problems — demonstrated that combining the NEAT approach with an ability to model local decision regions, results in the best performance to date on the most challenging task tested, i.e. half-field soccer with 50 start states. SNAP-NEAT, which attempts to determine the degree to which local mutations will be useful, was shown to be a competitive approach for these domains. These results provide evidence that SNAP-NEAT is a robust approach across a wide variety of problems.

Chapter 9

Discussion and Future Work

The work in this dissertation provides an explanation for NEAT's poor performance on fractured, high-level problems. The notion of fracture can be used both as a metric for measuring problem difficulty and as a tool for analyzing and improving learning algorithms. The results from this analysis — RBF-NEAT, Cascade-NEAT, and SNAP-NEAT – represent an important step towards developing the next generation of neuroevolution algorithms. There are also many questions raised by this analysis, however, as well as many interesting directions for future work.

9.1 Discussion of Results

Several of the experiments presented in Chapters 4 and 6 yielded interesting results, such as RBF-NEAT's frequent low performance and the synergy between SNAP-NEAT's modifications to the Adaptive Pursuit algorithm. This chapter begins with a discussion of these results.

9.1.1 The role of RBF-NEAT

For many of the problems analyzed in Chapter 6, Cascade-NEAT proves to be most effective at isolating local decision regions and generating high levels of performance. Given the dominance of these results, it seems worthwhile to examine the role of RBF-NEAT in learning fractured problems. Should RBF-NEAT ever be considered for such problems?

There are several reasons why RBF-NEAT should not be overlooked. First, the 10-Point classification problem from Chapter 6.1.1 is an example of a problem where RBF-NEAT yields the best performance. One explanation is the dimensionality of the state space; problems like N-Point classification have a relatively low input dimensionality compared to other problems. Adjusting the weights of basis functions is an effective way to model fracture in low-dimensional spaces, but it becomes infeasible as the dimensionality of the space increases. The experiments in maximizing variation from Chapter 6.3.1 support this theory; RBF-NEAT is able to generate the most variation for the 1-input case, but has increasing difficulty as the number of inputs increases.

Including RBF-NEAT as an option that algorithms like SNAP-NEAT can exercise allows these algorithms to perform well on fractured problems with relatively low dimensionality. The results for 10-Point classification (Figure 7.5) and variation maximization (Figure 7.17) show that SNAP-NEAT is able to perform well on these problems. Examining the operator probabilities learned in each case (Figures 7.6 and 7.18) shows that SNAP-NEAT relies heavily on the add-RBF-node operator when solving these problems. This result demonstrates that RBF-NEAT is useful, if only for limited scenarios. An interesting question for future work is to measure what performance penalty, if any, algorithms like SNAP-NEAT incur by including RBF-NEAT as a possible operator.

9.1.2 Modifications to Adaptive Pursuit

The generic Adaptive Pursuit algorithm described by Theirens makes several assumptions about the nature of the learning process that do not necessarily line up with the ideas behind NEAT (Thierens, 2005). In order to incorporate this Adaptive Pursuit algorithm into a NEAT-like algorithm more effectively, two changes were made: continuous updates and initial estimation. In Chapter 7.4, these modifications were examined independently to determine their effect on overall performance. Somewhat surprisingly, Figure 7.19 shows that when evaluated alone on the double pole-balancing problem, each of these these changes actually results in an overall decrease in performance.

On problems like the multiplexer, an analysis of these individual contributions shows a more expected result: continuous updates and initial estimation provide a moderate increase in performance over the baseline Adaptive Pursuit algorithm (Figure 7.21). Informal experiments performed when developing these modifications yielded results more similar to multiplexer than to pole-balancing, i.e. the modifications offered modest improvement when examined independently.

All of these results show that there is a synergy between continuous updates and initial estimation. On most problems, the net effect of both of these modifications is larger than the sum of their individual contributions. To some extent, this result is intuitive: When the operator values are initialized accurately, the best operators will tend to be chosen. If the correct operators for a problem offer more consistent feedback than the worst operators, using continuous updates will pull operator values in the right direction. Without accurate initialization, a poor choice of operators could result in noisy evaluations. Using continuous updates in this case could make the algorithm sensitive to noise early in the learning process, and run the risk of pulling operator values away from the correct distribution. On the other hand, using only initial estimation and avoiding continuous updates could set the

algorithm on the right path initially, but make it too slow at responding to changing operator values.

However, it is still surprising that the synergy between these two modifications exists to such an extent that the modifications actually lower performance when evaluated independently, as they do in Figure 7.19. It is possible that the dynamics of this problem — which are different from the other fractured problems analyzed, since they favor the NEAT algorithm — make these individual contributions more dependent on each other. It certainly seems reasonable that without a period of initial estimation, the continuous updates might use noisy data to change operator values too quickly, which could cause learning to diverge. It is less clear why initial estimation would only work when coupled with continuous updates. Further investigation of the interaction between these two modifications represents an interesting direction for future work.

9.2 Defining Fracture

The definition of fracture for algorithms such as NEAT was developed in Chapter 3 to measure problem difficulty. This section discusses the assumptions behind this definition, how it can be generalized, and how it relates to other metrics of difficulty.

9.2.1 Assumptions of total variation approach

The definition of fracture presented in Chapter 3 makes certain assumptions about both the learning algorithm and the problem being learned. This approach is useful in gaining insight into algorithms such as NEAT, which was the main focus of this dissertation. However, a broader definition of fracture could prove useful in the analysis of other algorithms and problems as well.

The definition of fracture as variation of optimal policies works when a learning algorithm incrementally builds up knowledge in a series of steps, each of which is

accompanied by a measurable increase in performance. This process might be akin to an algorithm initially always choosing the same action, and then gradually learning to choose different actions in certain situations after receiving positive rewards. As time passes, the amount of variation in solutions that this algorithm generates will increase, and its performance will improve. In this situation, higher levels of variation in optimal policies implies a greater "edit distance" between starting and ending policies. This increased distance in fractured problems should make it more difficult — or at least more time-consuming — for the algorithm to find solutions that perform well.

Constructive algorithms like NEAT fit this profile well. One of the core principles of NEAT is that learning should start with small networks, and expand those networks incrementally to more complicated forms. This process of complexification was designed to guide an initially smooth population of functions embodied in the networks to higher dimensionality, in which the functions could be made increasingly elaborate. Each mutation explored by NEAT is designed to add a small amount of variation to these functions, molding them from relatively smooth approximations to highly-detailed solutions. If the goal of learning is to find a highly-varied optimal policy, then NEAT will have to make a large number of small mutations, causing learning to proceed slowly.

As mentioned in Chapter 3.1, there are cases where this assumption does not hold. If there is not a significant difference in performance between initial policies and optimal policies, the amount of variation in optimal policies will not matter much. As an extreme example, consider a problem where the optimal policy has a high degree of variation, but where every other policy yields a score that is near that of the optimal policy. With this kind of pathological reward function, fracture is not a good predictor of problem difficulty because a learning algorithm does not have to be able to model fracture to perform well.

Even when there is a significant difference in performance between the initial and optimal policies, fracture may not measure difficulty for algorithms that do not have to make a long series of incremental changes. As an example, consider the task of approximating a high-frequency sine function. For the NEAT algorithm, this is a highly fractured task, since the optimal policy contains a high degree of variation and NEAT can only make small, incremental changes to network structure. However, if the learning algorithm has at its disposal the ability to generate a sine wave explicitly, then fitting all of the contours of the optimal wave can be accomplished quickly by tuning a few parameters. Because such a learning algorithm would not have to make a large number of incremental changes to reach an optimal policy, fracture would not gauge problem difficulty accurately.

The definition of fracture used in this dissertation was developed primarily to serve as a method for analyzing NEAT. Because NEAT develops networks in an incremental fashion, this analysis works well, and has led to the improved algorithms described above. This approach applies to other algorithms and problems to the degree to which these approaches and problems resemble those analyzed in this dissertation. For example, other constructive neuroevolution algorithms are likely to face similar problems, and would make fine candidates for similar analysis and improvements. Broadening this definition of fracture could make it more applicable to other areas, as well be discussed next.

9.2.2 Broadening the definition of fracture

An interesting direction for future work is to broaden the definition of fracture to include the number of changes that a learning algorithm must make to transform an initial policy into an optimal policy. This notion of "edit distance" between the starting point of learning and the desired endpoint could be formalized to take the operators used to move between those two points into account. Each edit might

correspond to a single operator being applied to a given policy, as opposed to a fixed notion of functional difference. Incorporating a search algorithm's operators into the definition of fracture could yield a more flexible and general definition of distance and of problem difficulty.

In a similar manner, it might be possible to address the problem of small score differentials by incorporating relative score into the fracture metric. For example, if the first policies explored performed at a certain level, then it could be possible to define difficulty in terms of the relative difference in performance between the initial and optimal policies. If the difference is small, it might be argued that the problem is not very fractured — simply because there is not much to gain from being able to model the fracture. Such an approach makes sense if the primary goal of such a fracture metric is to provide an absolute estimate of problem difficulty, instead of merely serving as a tool to help analyze the deficiencies of algorithms like NEAT.

However, there are further subtleties involving score differentials. For instance, some parts of the policy may affect score in a much more significant manner than others. For example, consider the function approximation problem shown in Figure 9.1. Instead of being judged on how closely a potential solution matches a target function (at every point equally), this function has certain "grey areas" where poor approximation is acceptable. While inside these grey areas, an approximation to this function is not severely punished for straying from the target function. Outside of the grey areas, however, the function must track the target precisely. On this example problem, the variation of the function inside the grey areas is of less importance, because the scoring metric is more lax there. It would be useful — albeit difficult — for a fracture metric to take such changes in the scoring metric into account.

One possible way of dealing with a variable scoring metric exists in a classic reinforcement learning scenario, where the goal is to learn a value function over states

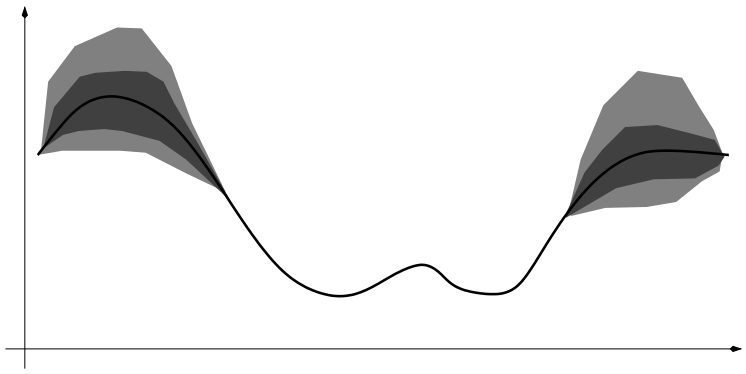


Figure 9.1: An example of a function approximation problem with a non-uniform scoring metric. The shaded areas near the function represent regions where deviations from the target function are acceptable. A fracture metric that can take such a variable scoring metric into account would yield a more accurate estimate of the difficulty of this problem.

and actions. Instead of considering fracture as a measure of variation in optimal policies, fracture could be defined as the variation of the optimal value functions for the problem. Such an approach would be beneficial in problems where two (or more) value functions are closely intertwined, changing value repeatedly but never by more than a small amount. An optimal policy for such a problem could be quite fractured if the two value functions vary out of phase with each other. In one sense, finding this optimal policy could be difficult, because it would be highly fractured. However, since the actual values for the value functions are similar, there would be few consequences for choosing a non-optimal action — in other words, a learning algorithm could perform well with a sub-optimal policy on this kind of problem. By incorporating value into fracture in this manner, this kind of problem would be considered to be unfractured.

It is interesting to note that the definition of fracture used in this dissertation is sensitive to scale. By simply measuring the amount of variation in an optimal policy, scale is effectively ignored, meaning that two identical policies that differ only in a multiplicative or constant factor would have differing amounts of fracture.

This assumption has proven useful for the analyses presented in this dissertation, but it is not always the right approach. More generally, it would make sense to consider two policies that differ only in scale equally difficult to learn. If the function approximator being used is completely scale-invariant, the fracture estimate should be scale-invariant as well: From the point of view of the function approximator, both problems are identical. However, not all function approximators are scale-invariant. A number of factors can determine how function approximators behave, including the range of values used to initialize them and "step size" that governs how quickly they can adjust their state. For such function approximators, learning values that are arbitrarily large could require numerous changes to internal state, which could be time-consuming and difficult. Incorporating a notion of scale into this definition of fracture and analyzing its effect with different learning algorithms is an interesting direction for future work.

9.2.3 Other measures of difficulty

Fracture is not the only way in which a problem could be considered difficult. As noted in Chapter 3, a number of related approaches have been employed (primarily for supervised learning problems) such as measuring the linearity of the decision boundary between two classes of points (Ho and Basu, 2002). Results in this dissertation, like those shown in Figure 4.6, also illustrate this phenomenon: Although performance tends to decrease as the variation of optimal policies increases, the correlation is not perfect. Some functions in that graph (e.g. those with variation near 12) are easier to solve than those with less variation (e.g. the function with variation near 10). This observation supports the conclusion that fracture is only one of several dimensions along which problems can be difficult to solve.

One concept that is closely related to variation is inflection. As the number of inflection points in a function increases, the function might be more difficult to

approximate. The results from the N-Points problems in Chapter 4.1.1 support this hypothesis. Increasing the separation parameter between points, v, was a natural way to increase variation. However, increasing the number of alternating points, N, caused performance to fall even when variation remained relatively constant (Figure 4.2). Even though variation encompasses inflection in some sense (all other things being equal, an increase in inflection will cause an increase in variation) a metric that includes inflection in its estimate of problem difficulty could be quite useful.

Another dimension along which problem difficulty might vary is repetition. Problems that have highly repetitive solutions might also have high degrees of variation, but be conceptually easy to solve. The sine-function example presented above demonstrates this dimension of difficulty. If a solution to a problem consists of a simple sine function, it could have a high degree of variation. However, if one of the tools that a learning algorithm has available to it is the ability to model a sine function explicitly, then solving this kind of problem would be simple. Another example might be a solution to a problem that repeats the same behavior in many different locations. An algorithm that has the ability to model this repetition would not have much difficulty in solving the problem.

It is intuitive that fracture is but one of many ways in which a problem can be difficult. How fracture relates to these various axes of difficulty is an important question for the future.

9.3 Directions for Future Work

In almost any scientific endeavor, answering one question will raise several more. The work presented in this dissertation is no exception – although it proposes and evaluates answers to several questions, there are many interesting avenues for future work.

9.3.1 Predicting fracture

One interesting use for a fracture metric would be to predict the difficulty of problems where the optimal solution is not known. While not presented in this dissertation, several informal experiments were performed that use a learning algorithm as a probe to estimate problem difficulty. The results suggest that such an approach might be feasible.

Figure 9.2 shows 25 runs of NEAT and Cascade-NEAT on a challenging sine-function approximation problem. Each point represents the score of a solution found during learning and the variation of that solution. The optimal solution to this problem is also shown. Even though Cascade-NEAT is not able to find solutions that are near optimal, the data that it generates over repeated runs suggests that it might be possible to estimate the variation of an optimal solution (assuming it was not already known). A best-fit linear regression over the data generates a line that predicts a variation estimate for any given score. If an experimenter desired to know the variation of an optimal policy (i.e. a policy with a score of 1.0) they could use this linear prediction to get a variation estimate that would be surprisingly close to the actual variation of the optimal policy.

This approach has been evaluated on several problems, to varying degrees of success. In some cases — like in Figure 9.2 — a linear extrapolation of learned data yields an excellent prediction of optimal variation. In other problems, such predictions produce less accurate results. Further understanding the merits of this approach is an interesting direction for future work.

9.3.2 Fracture with other algorithms

The idea of measuring the amount of information in a target solution is a general concept that could be applied to other learning paradigms such as supervised learning or reinforcement learning. As discussed above, the effectiveness of this metric

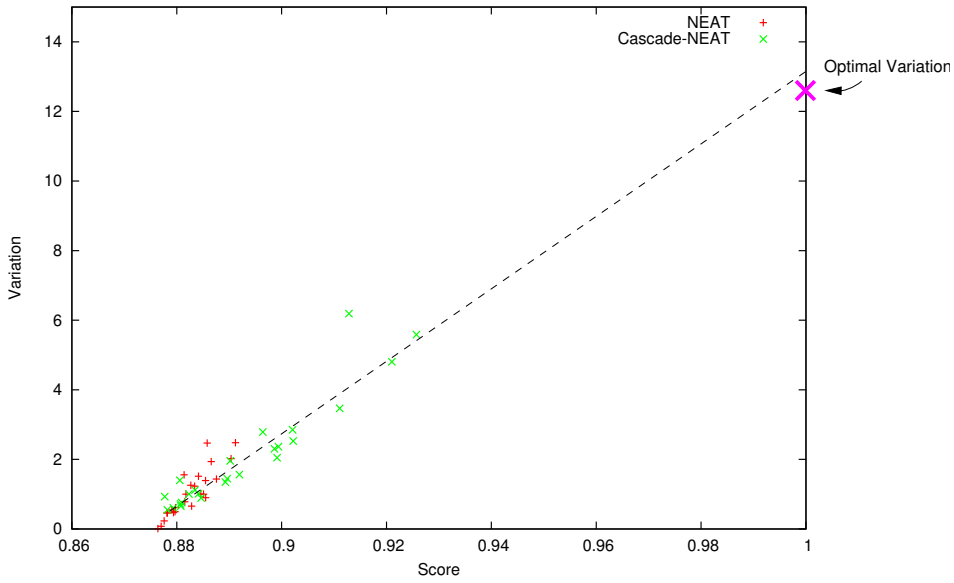


Figure 9.2: An example of how ongoing performance of a learning algorithm might be used to estimate the degree to which a problem is fractured. Shown are the final scores and variation measurements for 25 runs of NEAT and Cascade-NEAT on a sine-function approximation problem, as well as the actual optimal variation (which is known for this particular problem). Although Cascade-NEAT does not get a perfect score of 1.0 on this problem, the data that it generates might be useful in estimating how much variation a perfect solution might have. For instance, linear regression, shown as a line, predicts that variation of 12.9 is required when the optimal variation is actually 12.5. Such an estimate would be a useful predictor of problem difficulty.

depends on the presence of a significant performance and edit distance between initial and final solutions. To the extent that this is true, it should be possible to measure at least one aspect of a problem's difficulty using this definition of fracture.

When applying this metric to supervised learning, one interesting problem will be to determine how to convert a measurement of continuous functions to a measurement over discrete classes. In this work, the output of a neural network is interpreted as a multi-dimensional continuous function, even in cases when it might be interpreted as a discrete function. For example, when training players for the half-field offense problem, each player learns a mapping from state to one of several discrete actions. This discrete function is converted to a continuous function by taking an argmax over several continuous functions. Such an approach is common when working with neuroevolution algorithms, which must embody all of their solutions as continuous functions in neural networks. Not coincidentally, this approach also fits naturally with the metric of measuring fracture via function variation. However, it may not always be possible or useful to convert a discrete classification problem into a continuous problem. When faced with a supervised learning task involving discrete classes, it will be important to determine a useful way to measure variation.

The concept of measuring the amount of variation in a function is broadly applicable to almost any problem that involves function approximation. Applying this approach to reinforcement learning is another interesting direction for future work. Indeed, many of the ideas from this dissertation should be directly applicable to value-function approximation with neural networks. Other methods of function approximation could yield different results, depending on the degree to which they behave similarly to neural networks.

9.3.3 Modifications to SNAP-NEAT

Chapter 7 introduced SNAP-NEAT as an example of an adaptive operator selection algorithm. This class of algorithms explores how to best employ a set of operators during the learning process, usually making few or no assumptions about the nature of those operators. The operator selection mechanisms are relatively independent of the actual operators available, perhaps with the exception that performance could suffer from poor sampling if the number of operators becomes too large.

One interesting avenue for future work involves examining the role of operators other than add-node, add-link, add-cascade-node, and add-RBF-node. There are many different types of network topologies that have been explored in the neural network literature: networks with varying numbers of hidden layers, with or without recurrency, receptive fields that model those found in biology, etc. Also of interest are various fixed-topology approaches, like the multiple neuron population approach used by ESP (Gomez et al., 2006). A large array of operators inspired by these various topologies and organizational principles could provide an excellent base for an algorithm like SNAP-NEAT, allowing it to be applied to a broad spectrum of problems.

It is also interesting to note that on instances of several different problems (e.g. sine-function approximation) SNAP-NEAT is able to actually out-perform its constituent algorithms, making it the best performer of the lot. This result suggests that the best strategy for some problems involves the application of multiple operators. Since SNAP-NEAT (like Adaptive Pursuit on which it is based) is designed to heavily exploit the best performing operator, it may be possible to improve SNAP-NEAT's performance significantly by allowing it to explore combinations of multiple operators. Determining what kinds of problems might benefit from such a mix is also an interesting and challenging avenue for future work.

Improving the accuracy of SNAP-NEAT's operator value estimation would

certainly help it make better decisions about which operators to choose. As demonstrated by results for the non-Markov pole balancing problem (Figure 7.4), SNAP-NEAT does not always discover the best operators for a given problem. The modifications of continuous evaluations and an initial estimation period were shown to improve SNAP-NEAT's ability to estimate operator values correctly, but further enhancements may also be useful. For example, one possibility is to run multiple independent learning instances, each of which has a fixed association with one or more operators. By avoiding the application of different operators in succession, the individual merit of an operator or set of operators may be more clear. Incorporating ideas such as this into SNAP-NEAT's operator estimation code is an interesting direction for future work.

9.3.4 Constructing networks

As was discussed in Section 9.1.1, the results in this dissertation suggest that RBF-NEAT works best in low-dimensional settings. This result is understandable — as the number of inputs increases, the curse of dimensionality makes it increasingly difficult to set all of the parameters correctly for each basis function. This limitation suggests that a better method of incorporating basis functions into a constructive algorithm would be to situate those basis nodes on top of the evolved network structure. The lower levels of such a network can be thought of as transforming the input into a high-level representation, similar to the kernel transformation used by support vector machines (Boser et al., 1992). The high-level representation is likely to be of smaller dimensionality than the original representation and basis nodes operating at this level may be effective at selecting useful features. Determining how to evolve such a multi-stage network effectively would be an interesting direction for future work.

In addition to the cascade architecture and basis functions, there are other

useful ideas from the machine learning community that could be applied to neuroevolution. Chief among these possibilities is the potential for an initial unsupervised training period to initialize a large network, similar to the initial step of training that happens in deep learning. Using unsupervised learning to provide a good starting point for the search process could have a dramatic effect on learning performance.

9.3.5 Further evaluation of SNAP-NEAT

The data presented in this dissertation were drawn from a variety of different problems, ranging from simple but easy-to-analyze toy domains to challenging high-level strategy problems. The goal in examining such a broad spectrum of different problems was to obtain solid empirical evidence in support of the hypotheses, i.e. that

1. NEAT has difficulty on highly-fractured problems;
2. At least part of that difficulty stems from an inability to form local decision regions; and
3. This problem can be addressed through a judicious application of bias and constraint.

This empirical examination of NEAT's weaknesses has proven useful, yielding algorithms like SNAP-NEAT that can improve NEAT's performance on a variety of problems. However, algorithms like SNAP-NEAT can be applied to a large set of problems, and the current analysis only scratches the surface. There are countless challenging and interesting problems that learning algorithms currently can not solve, and the exploration of any of these problems could yield valuable insight into the strengths and weaknesses of algorithms like SNAP-NEAT. As discussed in Section 9.2.3, there are undoubtably many ways in which a problem might be considered difficult; empirical evaluation like the kind presented in this dissertation

is one of the most direct ways to identify these axes of difficulty and to determine which problems feature them.

In particular, it would be useful to evaluate the lessons learned in this dissertation on other high-level reinforcement learning problems. One potential candidate is a multi-agent vehicle control task, such as that examined in (Stanley et al., 2005a). Previous work showed that algorithms like NEAT are effective at generating low-level control behaviors, like efficiently steering a car through S-curves on a track. Evolving higher-level behavior to reason about opponents or race strategy has proven difficult, but may be possible with algorithms like Cascade-NEAT, RBF-NEAT, and SNAP-NEAT.

9.4 Conclusion

Many concepts explored in this dissertation — such as the definition of fracture and how it might be generalized or applied to other approaches to learning — represent interesting challenges could form the basis for significant research projects. This work provides many interesting starting points for future work in defining what makes problems hard, analyzing learning algorithms, and building the next generation of constructive neuroevolution algorithms.

Chapter 10

Conclusion

The work in this dissertation was inspired by the observation that while the NEAT neuroevolution algorithm works well on a variety of problems, it has particular difficulty in learning high-level strategic behavior. A series of experiments were described that investigate the hypothesis that this difficulty arises because these problems are fractured: The correct action varies discontinuously as the agent moves from state to state. In this chapter, the contributions of the dissertation are reviewed and expected impact of the work is estimated.

10.1 Contributions

In Chapter 3, a precise definition of problem fracture was provided based on the concept of function variation. If an optimal policy for a problem is known, then that policy can be treated as a continuous function and measured for variation. The resulting number is an indicator of the degree to which the problem is fractured.

Armed with a precise metric for measuring fracture, it became possible in Chapter 4 to examine the performance of NEAT on a variety of problems that vary in the degree to which they are fractured. Results from a set of experiments showed

that NEAT does in general perform at a lower level as fracture increases, confirming the fractured problem hypothesis. Further analysis, however, showed that NEAT does not always fail even in such problems: In rare situations it is able to find reasonable solutions. It turns out that by chance these solutions are formed via a series of local modifications. The conclusion is that NEAT performs poorly on fractured problems because it has difficulty in creating and manipulating such local decision regions systematically.

Fortunately, there are many algorithms from the related machine learning literature that address the problem of forming local decision regions. These approaches, described in Chapter 5, serve as inspiration for two new neuroevolution algorithms based on NEAT: RBF-NEAT, which augments the standard NEAT algorithm with the ability to add basis function nodes to network topologies, and Cascade-NEAT, which constrains the growth of network topologies to a specific and regular pattern. These two algorithms are designed to harness the power of bias and constraint in forming local decision regions to solve fractured problems.

An empirical comparison of RBF-NEAT and Cascade-NEAT to the standard NEAT algorithm in Chapter 6 showed that it is possible to increase performance on fractured problems significantly by employing such bias and constraint. However, further experimentation showed that NEAT still performs the best on traditional benchmark problems like pole-balancing.

Therefore, it is useful to incorporate RBF-NEAT, Cascade-NEAT, and regular NEAT into a single algorithm that can take advantage of the strengths of all three approaches. In Chapter 7, SNAP-NEAT was introduced as a modification of a popular adaptive operator selection algorithm and was shown empirically to work well on both fractured and unfractured problems. Further empirical evaluation in Chapter 8 showed that RBF-NEAT, Cascade-NEAT, and SNAP-NEAT work well not only on a benchmark suite of fractured problems but also on the challenging

high-level decision problems of keepaway and half-field soccer.

10.2 Conclusion

Continued improvement in state-of-the-art artificial intelligence algorithms requires more than just exploring what kinds of problems current algorithms can solve — it is vital to understand why the algorithms have difficulty on certain problems. This dissertation presents an empirical investigation of why one popular neuroevolution algorithm, NEAT, has difficulty in domains that require high-level strategy, and how this difficulty can be overcome through judicious use of bias and constraint. The methods used to analyze NEAT and the resulting algorithms offer many interesting directions for future work, from a continued empirical analysis of neuroevolution to applications of these approaches to other learning algorithms and problems.

Furthermore, this dissertation shows how neuroevolution can be scaled up from learning reactive control to problems where high-level strategy is required. Such problems are common in the real world, ranging from an intelligent division of labor among a group of robots cleaning a hazardous waste site, to software that helps an autonomous vehicle navigate through a crowded urban environment. These applications of learning algorithms to difficult, high-level learning problems should allow machine learning in general and neuroevolution in particular to have high impact in the future.

Bibliography

Angeline, P. J. (1997). Evolving basis functions with dynamic receptive fields. In *IEEE International Conference on Systems, Man, and Cybernetics*, vol. 5, 4109–4114.

Angeline, P. J., Saunders, G. M., and Pollack, J. B. (1993). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5:54–65.

Asada, M., Noda, S., and Hosoda, K. (1995). Non-physical intervention in robot learning based on lfe method. In *Proceedings of Machine Learning Conference Workshop on Learning from Examples vs. Programming by Demonstration*, 25–31.

Barbosa, H. J. C., and Sa, A. M. (2000). On adaptive operator probabilities in real coded genetic algorithms. In *In Proc. XX International Conference of the Chilean Computer Science Society*.

Barron, A., Rissanen, J., and Yu, B. (1998). The minimum description length principle in coding and modeling. *IEEE Trans. Information Theory*, 44(6):2743–2760.

Bengio, Y. (2007). Learning deep architectures for ai. Technical Report 1312, Dept. IRO, Universite de Montreal.

Billings, S. A., and Zheng, G. L. (1995). Radial basis function network configuration using genetic algorithms. *Neural Networks*, 8:877–890.

Bochner, S. (1959). *Lectures on Fourier Integrals with an Author's Supplement on Monotonic Functions, Stieltjes Integrals and Harmonic Analysis*. Princeton University Press.

Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, 144–152. New York, NY, USA: ACM.

Bull, L., and O'Hara, T. (2002). Accuracy-based neuro and neuro-fuzzy classifier systems. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 905–911.

Butz, M. V. (2005). Kernel-based, ellipsoidal conditions in the real-valued xcs classifier system. In *Proceedings of the 2005 conference on Genetic and Evolutionary Computation*, 1835–1842.

Butz, M. V., and Herbort, O. (2008). Context-dependent predictions and cognitive arm control with xcsf. In *Proceedings of the 2008 conference on Genetic and Evolutionary Computation*.

Chaitin, G. (1975). A theory of program size formally identical to information theory. *Journal of the ACM*, 22:329–340.

Chaiyaratana, N., and Zalzala, A. M. S. (1998). Evolving hybrid rbf-mlp networks using combinedgenetic/unsupervised/supervised learning. In *UKACC International Conference on Control*, vol. 1, 330–335.

DaCosta, L., Fialho, A., Schoenauer, M., and Sebag, M. (2008). Adaptive operator selection with dynamic multi-armed bandits. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, 913–920.

Dasgupta, D., and McGregor, D. (1992). Designing application-specific neural networks using the structured genetic algorithm. In *In Proceedings of the International Conference on Combinations of Genetic Algorithms and Neural Networks*, 87–96.

Davis, L. (1989). Adapting operator probabilities in genetic algorithms. In *Proc. 3rd International Conference on Genetic Algorithms*, 61–69.

Fahlman, S. E., and Lebiere, C. (1990). The cascade-correlation learning architecture. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems*, vol. 2, 524–532. Denver 1989: Morgan Kaufmann, San Mateo.

Ghosh, J., and Nag, A. (2001). An overview of radial basis function networks. *Studies In Fuzziness And Soft Computing: Radial basis function networks 2: new advances in design*, 1–36.

Goldberg, D. (1990). Probability matching, the magnitude of reinforcement, and classifier system bidding. 5(4):407–426.

Goldberg, D. E., and Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optimization. In *Proceedings of the Second International Conference on Genetic Algorithms*, 148–154.

Gomez, F. (2003). *Robust Non-Linear Control Through Neuroevolution*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin.

Gomez, F., and Miikkulainen, R. (1999). Solving non-markovian control tasks with neuroevolution. In *In Proceedings of the 16th International Joint Conference on Artificial Intelligence*.

Gomez, F., and Miikkulainen, R. (2003). Active guidance for a finless rocket using neuroevolution. In *In Proceedings of the Genetic and Evolutionary Computation Conference*.

Gomez, F., and Miikkulainen, R. (2004). Transfer of neuroevolved controllers in unstable domains. In *In Proceedings of the Genetic and Evolutionary Computation Conference*.

Gomez, F., Schmidhuber, J., and Miikkulainen, R. (2006). Efficient non-linear control through neuroevolution. In *Proceedings of the European Conference on Machine Learning (ECML-06, Berlin)*.

Gonzalez, J., Rojas, I., Ortega, J., Pomares, H., Fernandez, F., and Diaz, A. (2003). Multiobjective evolutionary optimization of the size, shape, and position parameters of radial basis function networks for function approximation. *IEEE Transactions on Neural Networks*, 14:1478–1495.

Gruau, F., Whitley, D., and Pyeatt, L. (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, 81–89. MIT Press.

Guillen, A., Pomares, H., Gonzalez, J., Rojas, I., Herrera, L. J., and Prieto, A. (2007). Parallel multi-objective memetic rbfnns design and feature selection for function approximation problems. 4507/2007:341–350.

Guillen, A., Rojas, I., Gonzalez, J., Pomares, H., Herrera, L. J., and Paechter, B. (2006). Improving the performance of multi-objective genetic algorithm for function approximation through parallel islands specialisation. 4304/2006:1127–1132.

Guo, L., Huang, D.-S., and Zhao, W. (2003). Combining genetic optimisation with hybrid learning algorithm for radial basis function neural networks. *Electronics Letters*, 39:1600–1601.

Gutmann, H. (2001). A radial basis function method for global optimization. *Journal of Global Optimization*, 19:201–227.

Hinton, G. E., and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507.

Ho, T., and Basu, M. (2002). Complexity measures of supervised classification problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(3):289–300.

Hornby, G., and Pollack, J. (2002). Creating high-level components with a generative representation for body-brain evolution. *Artificial Life*, 8.

Hornik, K. M., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 359–366.

Howard, D., Bull, L., and Lanzi, P.-L. (2008). Self-adaptive constructivism in neural xcs and xcsf. In *Proceedings of the 2008 Genetic and Evolutionary Computation Conference*.

Igel, C. (2003). Neuroevolution for reinforcement learning using evolution strategies. In *Congress on Evolutionary Computation 2003 (CEC 2003)*.

Jiang, N., Zhao, Z., and Ren, L. (2003). Design of structural modular neural networks with genetic algorithms. *Advances in Software Engineering*, 34:17–24.

Julstrom, B. A. (1995). What have you done for me lately? adapting operator probabilities in a steady-state genetic algorithm. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, 81–87.

Kaelbling, L. P. (1993). *Learning in Embedded Systems*. MIT Press.

Kalyanakrishnan, S., Liu, Y., and Stone, P. (2007). Half field offense in RoboCup soccer: A multiagent reinforcement learning case study. In Lakemeyer, G., Sklar, E., Sorenti, D., and Takahashi, T., editors, *RoboCup-2006: Robot Soccer World Cup X*, 72–85. Berlin: Springer Verlag.

Kamke, E. (1956). Das lebesgue-stieltjes integral.

Kohl, N., Stanley, K., Miikkulainen, R., Samples, M., and Sherony, R. (2006). Evolving a real-world vehicle warning system. In *Proceedings of the Genetic and Evolutionary Computation Conference 2006*, 1681–1688.

Kolmogorov, A. (1965). Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:4–7.

Kretchmar, R., and Anderson, C. (1997). Comparison of cmacs and radial basis functions for local function approximators in reinforcement learning. In *Proceedings of the International Conference on Neural Networks*.

Lanzi, P., Loiacono, D., Wilson, S., and Goldberg, D. (2006). Classifier prediction based on tile coding. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 1497–1504.

Lanzi, P. L., Loiacono, D., Wilson, S. W., and Goldberg, D. E. (2005). Xcs with computed prediction for the learning of boolean functions. In *Proceedings of the IEEE Congress on Evolutionary Computation Conference*.

Lawrence, S., Tsoi, A., and Back, A. (1996). Function approximation with neural networks and local methods: Bias, variance and smoothness. In *Australian Conference on Neural Networks*, 16–21.

LeCun, Y., and Bengio, Y. (2007). Scaling learning algorithms towards ai. *Large-Scale Kernel Machines*.

Leonov, A. S. (1998). On the total variation for functions of several variables and a multidimensional analog of helly's selection prinicple. *Mathematical Notes*, 63(1):61–71.

Li, J., and Duckett, T. (2005). Q-learning with a growing rbf network for behavior learning in mobile robotics. In *Proceedings of the Sixth IASTED International Conference on Robotics and Applications*.

Li, J., Martinez-Maron, T., Lilienthal, A., and Duckett, T. (2006). Q-ran: A constructive reinforcement learning approach for robot behavior learning. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robot and System*.

Li, M., and Vitanyi, P. (1993). *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag.

Maciejowskia, J. M. (1979). Model discrimination using an algorithmic information criterion. *Automatica*, 15:579–593.

Maillard, E., and Gueriot, D. (1997). Rbf neural network, basis functions and genetic algorithm. In *International Conference on Neural Networks*, vol. 4.

Mitchell, T. (1997). *Machine Learning*. McGraw Hill.

Moody, J., and Darken, C. J. (1989). Fast learning in networks of locally tuned processing units. *Neural Computation*, 1:281–294.

Moriarty, D. E., and Miikkulainen, R. (1996). Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22:11–32.

Park, J., and Sandberg, I. W. (1991). Universal approximation using radial-basis-function networks. *Neural Computation*, 3:246–257.

Peterson, T., and Sun, R. (1998). An rbf network alternative for a hybrid architecture. In *IEEE International Joint Conference on Neural Networks*, vol. 1, 768–773.

Platt, J. (1991). A resource-allocating network for function interpolation. *Neural Computation*, 3(2):213–225.

Potter, M. A., and Jong, K. A. D. (2000). Cooperative coeveolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29.

Pujol, J. C. F., and Poli, R. (1997). Evolution of the topology and the weights of neural networks using genetic programming with a dual representation. Technical Report CSRP-97-7, School of Computer Science, The University of Birmingham, Birmingham B15 2TT, UK.

Radcliffe, N. J. (1993). Genetic set recombination and its application to neural network topology optimization. *Neural Computing and Applications*, 1(1):67–90.

Reisinger, J., Bahceci, E., Karpov, I., and Miikkulainen, R. (2007). Coevolving strategies for general game playing. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*.

Rosca, J. P. (1997). *Hierarchical Learning with Procedural Abstraction Mechanisms*. PhD thesis, Rochester, NY 14627, USA.

Saravanan, N., and Fogel, D. B. (1995). Evolving neural control systems. *IEEE Expert*, 23–27.

Sarimveis, H., Alexandridis, A., Mazarakis, S., and Bafas, G. (2004). A new algorithm for developing dynamic radial basis function neural network models based on genetic algorithms. *Computers and Chemical Engineering*, 28:209–217.

Shilov, G. E., and Gurevich, B. L. (1967). Integral, measure, derivative.

Stanley, K., Kohl, N., Sherony, R., and Miikkulainen, R. (2005a). Neuroevolution of an automobile crash warning system. In *Proceedings of the Genetic and Evolutionary Computation Conference 2005*, 1977–1984.

Stanley, K., and Miikkulainen, R. (2003). A taxonomy for artificial embryology. *Artificial Life*, 9:93–130.

Stanley, K. O. (2003). *Efficient Evolution of Neural Networks Through Complexification*. PhD thesis, Department of Computer Sciences, University of Texas at Austin.

Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2005b). Real-time neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation*, 9(6):653–668.

Stanley, K. O., D'Ambrosio, D. B., and Gauci, J. (2009). A hypercube-based indirect encoding for evolving large-scale neural networks. *Artificial Life*.

Stanley, K. O., and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2).

Stanley, K. O., and Miikkulainen, R. (2004a). Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100.

Stanley, K. O., and Miikkulainen, R. (2004b). Evolving a roving eye for go. In *Proceedings of the Genetic and Evolutionary Computation Conference*.

Stone, P., Kuhlmann, G., Taylor, M. E., and Liu, Y. (2006). Keepaway soccer: From machine learning testbed to benchmark. In Noda, I., Jacoff, A., Bredenfeld, A., and Takahashi, Y., editors, *RoboCup-2005: Robot Soccer World Cup IX*, vol. 4020, 93–105. Berlin: Springer Verlag.

Stone, P., Sutton, R. S., and Kuhlmann, G. (2005). Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, 1038–1044.

Sutton, R. S., and Barto, A. G. (1998). *Reinforcement Learning I: Introduction*.

Taylor, M., Whiteson, S., and Stone, P. (2006). Comparing evolutionary and temporal difference methods for reinforcement learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 1321–28.

Thierens, D. (2005). An adaptive pursuit strategy for allocating operator probabilities. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, 1539–1546.

Tulai, A. F., and Oppacher, F. (2002). Combining competitive and cooperative coevolution for training cascade neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 618–625.

Tuson, A., and Ross, P. (1998). Adapting operator settings in genetic algorithms. *Evolutionary Computation*, 6(2):161–184.

Vapnik, V., and Chervonenkis, A. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16:264–280.

Vitushkin, A. G. (1955). On multidimensional variations.

Wedge, D., Ingram, D., McLean, D., Mingham, C., and Bandar, Z. (2005). Neural network architectures and wave overtopping. In *Proc. Inst. Civil Engineering 2005: Maritime Engineering*, vol. 158 MA3, 123–133.

Wedge, D., Ingram, D., McLean, D., Mingham, C., and Bandar, Z. (2006). On global-local artificial neural networks for function approximation. *IEEE Transactions on Neural Networks*, 17(4):942–952.

Whitehead, B., and Choate, T. (1996). Cooperative-competitive genetic evolu-

tion of radial basis functioncenters and widths for time series prediction. *IEEE Transactions on Neural Networks*, 7:869–880.

Whiteson, S., Kohl, N., Miikkulainen, R., and Stone, P. (2005). Evolving keepaway soccer players through task decomposition. *Machine Learning*, 59:5–30.

Whitley, D., Dominic, S., Das, R., and Anderson, C. W. (1993). Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, 13:259–284.

Wieland, A. (1991). Evolving neural network controllers for unstable systems. In *In Proceedings of the International Joint Conference on Neural Networks*, 667–673.

Wilson, S. W. (2002). Classifiers that approximate functions. *Natural Computing*, 1:211–234.

Wilson, S. W. (2008). Classifier conditions using gene expression programming. Technical Report 2008001, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447.

Yong, C. H., and Miikkulainen, R. (2007). Coevolution of role-based cooperation in multi-agent systems. Technical Report AI07-338, Department of Computer Sciences, University of Texas at Austin.

Vita

Nate Kohl attended Carleton College, graduated in 2000, and worked for a year before enrolling in graduate school at the University of Texas at Austin.

Permanent Address: Department of Computer Sciences
1 University Station
C0500
Austin, TX 78712

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.